VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Systém pro sběr a zpracování chyb z koncových zařízení

Endpoint Device Error Log Collection and Processing System

Bc. Juraj Kubala

Diplomová práce

Vedoucí práce: Ing. David Ježek Ph.D.

Ostrava, 2021

Abstrakt

Táto diplomová práca sa zaoberá problematikou zaznamenávania chybových hlásení v aplikáciách. Pozornosť je upriamená hlavne na webové aplikácie, ktoré sa v posledných rokoch tešia veľkej popularite a rozmachu. Na začiatku sa v práci hovorí o dôležitosti zaznamenávania udalostí v aplikácií a ozrejmuje sa, čo by mal záznam obsahovať aby bol užitočný. Ďalej práca popisuje špecifické vlastnosti webových aplikácií a technológie, pomocou ktorých sú vyvíjané. V súčasnosti existujú nástroje, ktoré sa zaoberajú otázkou zaznamenávania udalosti v aplikáciách. Niektoré z týchto nástrojov sú v tejto práci predstavene a porovnané. Podstatná časť práce je venovaná návrhu vlastného nástroja a jeho implementácií. Čitateľ má možnosť dozvedieť sa o použitých technológiách a rozhodnutiach, ktoré boli prijaté vo fáze návrhu. Na záver je vykonané záverečné zhrnutie vrátane popisu funkcií a použiteľnosti vytvoreného nástroja.

Kľúčové slová: hlásenia o udalostiach, chybové hlásenia, webová aplikácia, JavaScript, Elasticsearch, Logstash, ReactJS, Angular

Abstract

This diploma thesis deals with the process of recording errors and other events in applications. Attention is focused mainly on web applications, which have risen great popularity and growth in recent years. At the beginning, the thesis talks about the importance of recording events in the application and clarifies what the record should contain in order to be useful. Furthermore, the thesis describes the specific features of web applications and technologies which are used for their development. There are existing tools for event logging. Some of these tools are described and compared in this thesis. A substantial part of the thesis is devoted to the design of custom tool for logging and its implementation. The reader has the opportunity to learn about the technologies used and about the decisions that have been taken during the design phase. Finally, a final summary is made, including a description of the tool and its usability.

Keywords: logs, error logs, web application, JavaScript, Elasticsearch, Logstash, ReactJS, Angular

Rád by som na tomto mieste poďakoval vedúcemu diplomovej práce pánu Ing. Davidovi Ježkovi Ph.D. za ľudský prístup, cenné rady a pripomienky pri tvorbe tejto práce.

Rád by som poďakoval aj mojej rodine, kamarátom a spolužiakom za podporu počas štúdia.

Obsah

Zoznam použitých symbolov a skratiek

CLI	- Command Line Interface
CRUD	- Create, Read, Update, Delete
CSS	- Cascading Style Sheets
DOM	- Document Object Model
DI	- Dependency injection
DTO	- Data Transfer Object
HTML	- Hypertext Markup Language
HTTP	- Hypertext Transfer Protocol
IoC	- Inversion of control
IP	- Internet Protocol
JPA	- Java Persistence API
JSON	- JavaScript Object Notation
MVC	- Model View Controller
OS	- Operating System
SOM	- Self-Organizing map
PWA	- Progressive web application
REST API	- Representational State Transfer Application Programming Interface
SEO	- Search Engine Optimization.
SSR	- Server-side rendering
SaaS	- Software-as-a-Service
URL	- Uniform Resource Locator
UML	- Unified Modelling Language
W3C	- World Wide Web Consortium

Zoznam ilustrácií a zoznam tabuliek

Zoznam tabuliek

1. Úvod

Vývoj softvéru je komplexný proces, ktorý zahŕňa rôzne fázy od definície problému, návrhu riešenia, implementácie až po testovanie a nasadenie. Je chybou predpokladať, že odovzdaním projektu zákazníkovi sa tento proces končí. Fáza údržby softvéru je naopak často niekoľkonásobne dlhšia. Nastavenie správnych procesov počas vývoja a testovania softvéru môže minimalizovať počet chýb vo výslednej aplikácii, nikdy však nie je možné tvrdiť, že softvér je bezchybný.

Keďže nie je možné eliminovať všetky potenciálne chyby, je možné z praktických skúseností tvrdiť, že skôr či neskôr príde čas, kedy sa ozve zákazník s tým, že sa v aplikácii niečo pokazilo. Samotnej oprave chyby však predchádza zisťovanie okolností, za ktorých k chybe došlo. Jedným zdrojom informácií môže byť zákazník. Informácie od zákazníka však nemusia byť presné a môžu byť skreslené jeho subjektívnym vnímaním situácie. Ďalším zdrojom informácií sú záznamy udalostí, ktoré by mala generovať samotná aplikácia. Analýza týchto záznamov nám poskytuje objektívny prehľad o udalostiach, ktoré nastali počas vzniku chyby. Tieto záznamy sú nenahraditeľnou pomôckou pri skúmaní vzniknutej chyby, neštandardného správania, ale aj pri ladení aplikácie.

V tejto diplomovej práci sa zameriavam hlavne na zaznamenávanie udalostí vo webových aplikáciách, ktoré sa v posledných rokoch tešia veľkej popularite a rozmachu. Vďaka moderným prehliadačom a technológiám ako React a Angular sa v klientskej časti aplikácie objavuje čoraz viac logiky a preto je potrebné takúto aplikáciu monitorovať. Na riešenie problému monitorovania existujú nástroje, ktoré uľahčujú správu a analýzu záznamov. Niektoré z týchto nástrojov budú v tejto práci predstavené a porovnané. Hlavným cieľom tejto práce je však návrh a implementácia vlastného riešenia pre záznam a správu hlásení z webových aplikácií.

V prvých kapitolách sa zameriavam na predstavenie problematiky zaznamenávania udalostí v aplikáciách a jej prínosoch. Je tu popísané, čo by mal záznam obsahovať, aby bol užitočný pre programátora alebo správcu systému. Následne sú popísané vlastnosti webových aplikácií, ich výhody aj nevýhody. Súčasťou tohto popisu je aj predstavenie technológií, ktoré sa dnes na ich tvorbu široko používajú a sú tak ich neoddeliteľnou súčasťou.

V praktickej časti tejto práce popisujem proces návrhu a prijaté rozhodnutia. Sú tu popísané aj technológie použité na implementáciu. Praktická časť pozostáva z niekoľkých častí, vrátane nastavenia platformy Elastic Stack, návrhu a vývoja serverovej aplikácie na správu záznamov udalostí a používateľského rozhrania. Časť práce je venovaná návrhu a implementácii knižnice pre webové aplikácie, ktorej hlavnou úlohou je zaznamenávať nielen chybové udalosti a odosielať ich na server, kde sú spracované a uložené.

V poslednej časti zhrniem výsledky tejto práce a užitočnosť vyvinutého nástroja. Taktiež sa tu zamýšľam nad ďalším vývojom nástroja a implementovaním funkcií, ktoré by rozšírili jeho použiteľnosť.

Vytvorený nástroj bude použitý ako interný systém na monitorovanie webových aplikácií vo firme Inmics Oy. Firma vyvíja rôzne aplikácie, hlavne s využitím technológií React a Angular. Jedna z nich slúži ako nástroj pri vykonávaní inšpekcií budov a iných priestorov. Inšpekcie sú vykonávané vo všetkých fázach životného cyklu monitorovanej budovy, a to vrátane prípravy pozemku, budovaniu

hrubej stavby, rozvodu technických sietí a inštalácie potrebných zariadení. Pri niektorých budovách prebiehajú inšpekcie aj po odovzdaní budovy jej vlastníkovi.

V súčasnosti tieto aplikácie neimplementujú žiadne nástroje, ktoré by zaznamenávali chybové udalosti, a tak ladenie a odstraňovanie vzniknutých chýb je veľmi náročné. Samotná doba na opravu chyby je často niekoľkonásobne kratšia v porovnaní s časom potrebným na zistenie dôvodu, prečo k chybe došlo.

Vyvíjaný nástroj však nebude šitý na mieru spomenutým aplikáciám, ale bude implementovať všeobecné princípy webových aplikácií tak, aby mohol byť použitý aj v iných aplikáciách.

2. Zaznamenávanie chybových hlásení

So slovom *logging, logovanie* alebo zbieranie a uchovávanie záznamov udalostí sa sterol pravdepodobne každý vývojár. V informatike sa slovom *log* označuje záznam o určitej, viac či menej významnej udalosti alebo činnosti. Jedná sa o základný prvok, ktorý je generovaný operačným systémom, zariadením ale aplikáciou ako reakcia na určitý stimul. Stimul, ktorý iniciuje vygenerovanie log záznamu silne závisí od prostredia v ktorom vzniká. Napríklad operačný systém vytvára záznam pri prihlásení a odhlásení používateľa, webový server zaznamenáva čo a kedy bolo pristúpené, a pevný disk môže vytvárať záznam ak nastane chyba pri čítaní z média.

Zaznamenávanie udalostí býva často podceňované, aj napriek skutočnosti, že práve tieto záznamy môžu byť zdrojom cenných informácií pri hľadaní a analyzovaní chýb v rôznych systémoch či aplikáciách. Ak v systéme nastala chyba, záznamy o chybe nám pomáhajú pri jej identifikácií a lepšiemu pochopeniu, za akých okolností nastala. Informácie, ktoré sa zaznamenávajú určuje tvorca aplikácie alebo zariadenia. Je vhodné nájsť rovnováhu medzi množstvom záznamov a prínosom danej informácie pri skúmaní uložených záznamov.

Nie všetky záznamy o udalostiach vznikajú ako reakcia na chybu alebo problém v aplikácií. Niektoré záznamy iba informujú o udalostiach, ktoré nastali v aplikácií alebo môžu slúžiť pri ladení algoritmov. Na základe významnosti rozdeľujeme tieto záznamy nasledovne [1]:

- INFO – Správy tohto typu sú určené na to, aby informovali vývojárov a správcov o tom, že nastala nejaká neškodná udalosť. Takéto správy môžu byť vygenerované napríklad, keď sa systém reštartuje. Ak však k reštartu dôjde mimo bežnej údržby alebo pracovnej doby, môže to byť dôvod na poplach.
- WARNING – Varovné správy sa týkajú situácie, keď v systéme chýbajú alebo sú potrebné veci, ktorých absencia však nebude mať vplyv na fungovanie systému. Napríklad ak programu nie je daný správny počet argumentov príkazového riadku, ale napriek tomu môže bežať bez nich. V takomto prípade môže program zobraziť varovanie pre používateľa. Ďalším príkladom môže byť upozornenie na používanie zastaralej knižnice alebo volanie zastaralej metódy.
- ERROR – Chybové správy sa používajú na prenos informácií o chybách, ktoré sa vyskytujú na rôznych úrovniach počítačového systému. Napríklad operačný systém môže vygenerovať chybový protokol, keď nemôže synchronizovať vyrovnávacie pamäte na disk. Mnohé chybové správy, bohužiaľ, poskytujú iba začiatočný bod, prečo k nim došlo. Často je potrebné vykonať ďalšie skúmanie, aby sme sa dostali k hlavnej príčine chyby.
- ALERT – Výstraha má znamenať, že sa stalo niečo zaujímavé. Varovania sú vo všeobecnosti doménou bezpečnostných zariadení a systémov súvisiacich so zabezpečením. Systém prevencie narušenia (IPS) môže byť nainštalovaný v počítačovej sieti a skúmať všetku prichádzajúcu komunikáciu. Na základe obsahu paketových údajov určí, či je dané sieťové pripojenie povolené. Ak IPS narazí na pripojenie, ktoré by mohlo byť škodlivé, môže vykonať ľubovoľný počet predkonfigurovaných akcií. Rozhodnutie sa spolu s vykonanou akciou zaznamená. [1]

Na to aby bol log záznam užitočný, mal by prinášať odpovede na nasledujúce otázky:

- Aká udalosť nastala?
- Kedy táto udalosť nastala?
- Kde udalosť nastala?
- Kto alebo čo vyvolal vznik udalosti?
- Aký objekt bol udalosťou dotknutý?

Zodpovednosť nájsť rovnováhu medzi veľkým množstvom neužitočných záznamov a chýbajúcim záznamom v prípade šetrenia závažnej situácie, leží často na pleciach samotného vývojára. To, ktoré udalosti sa rozhodne zaznamenávať, je spravidla v jeho rukách. Existujú však situácie, pri ktorých stojí za zváženie tieto udalosti zaznamenať:

- Manipulácia s existujúcim objektom. Ak upravujeme alebo mažeme objekt v aplikácií, stojí za zváženie túto udalosť zaznamenať. Do budúcnosti nám to ponúka možnosť sledovať históriu zmien objektu.
- Autentifikácia a autorizácia používateľa. Zaznamenávať tento typ udalosti nám dáva možnosť odhaliť únik prihlasovacích údajov používateľa, zaznamenať neobvyklý čas prihlásenia do aplikácie mimo pracovnú dobu alebo nájsť chybu v nastavení práv. Tento druh aktivity by mala zaznamenávať každá aplikácia.
- Štart, vypnutie alebo reštart aplikácie. Aplikácia môže byť reštartovaná pravidelne na základe predom stanoveného rozvrhu aby fungovala spoľahlivo. Môže sa však stať, že aplikácia bola reštartovaná mimo stanovený rozvrh, a preto je vhodné mať o tejto udalosti záznam.

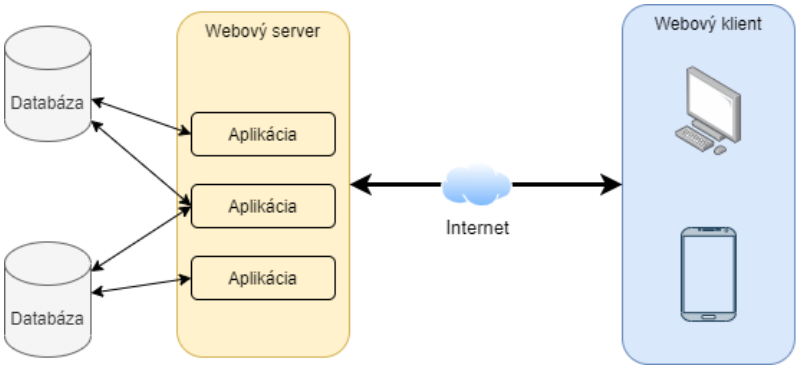
Podľa knihy *Logging and Log Management* by každý dobrý záznam o udalosti mal obsahovať nasledovné informácie:

- Dátum a čas vzniku udalosti
- Typ udalosti
- Identifikátor systému alebo aplikácie, v ktorej udalosť vznikla
- Indikátor, či sa jedná o úspešnú alebo neúspešnú udalosť
- Indikátor dôležitosti záznamu o udalosti
- Pri udalostiach spojených s používateľom, by mal záznam obsahovať jeho identifikátor [1]

3. Webové aplikácie

V dnešnej dobe navštevujeme rôzne webové stránky. Môže sa jednať o rôzne prezentačné stránky firiem a produktov, stránky spravodajských webov, rôzne blogy, sociálne siete alebo stránky s video obsahom. Hlavným cieľom webových stránok je poskytovať jednosmerný prístup k informáciám. Každá webová stránka je jednoznačne identifikovaná URL adresou. Pri webových stránkach sa môžeme stretnúť často s označením statické, čo napovedá tomu, že ich obsah sa dynamicky nemení. Dobrým príkladom statických webových stránok môžu byť blogy, spravodajské články alebo Wikipédia.

Programy boli tradične inštalované na osobné počítače z rôznych dátových nosičov. Rozvoj internetu a webových prehliadačov napomohol vzniku nového spôsobu vývoja a distribúcie aplikácií, a to webovým aplikáciám. Webové aplikácie, podobne ako statické webové stránky, sú pre používateľa zdrojom informácií. Oproti statickým stránkam však poskytujú rozšírenú funkcionalitu a dynamicky meniaci sa obsah. Príkladom môže by emailový klient Gmail alebo sociálna sieť Facebook. Internetové obchody môžeme takisto zaradiť medzi webové aplikácie, keďže okrem informáciách o produkte poskytujú možnosť platby za tovar, sledovanie zásielok a iné.



Obr. 1: Princíp webovej aplikácie

Webové aplikácie sa stali populárnymi vďaka všadeprítomnému webovému prehliadaču. Používateľ môže webovú aplikáciu používať okamžite, bez nutnosti jej inštalácie na svoj počítač. Nespornou výhodou webových aplikácií je rýchle doručenie nových vydaní aplikácie za účelom opravy chýb alebo implementácie nových vlastností a funkcií. Ďalšou z výhod je možnosť používať webové aplikácie na rôznych operačných systémoch a platformách. Tvorcovia webových aplikácií tak môžu tvoriť aplikácie bez nutnosti znalosti špecifického programovacieho jazyka pre daný operačný systém. Na to, aby sa webové aplikácie zobrazovali a fungovali správne, musia tvorcovia dodržiavať postupy a štandardy, ktoré boli vytvorené pre tvorbu webových stránok a aplikácií. Multiplatformovú podporu potom zabezpečujú tvorcovia webových prehliadačov.

Na zastrešenie technických špecifikácií štandardov pre webové stánky a aplikácie vznikla organizácia World Wide Web Consortium (W3C). Je to organizácia, ktorá zjednocuje a tvorí technické špecifikácie a odporúčania pre vývojárov s cieľom zjednotiť technický základ pre technológie, ktoré sú použité na webe. [2]

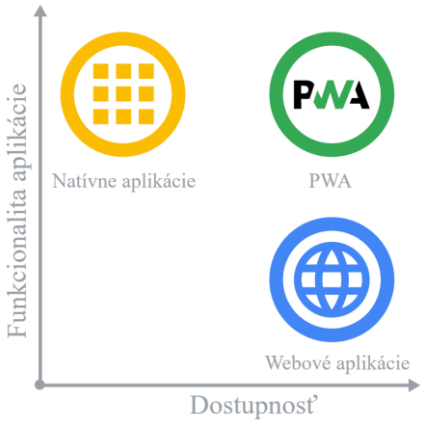
Webové aplikácia neprinášajú so sebou iba pozitíva ale aj pár negatív. Jedna z výziev, ktorá pri vývoji vzniká je spojená s jednou z hlavných výhod webovej aplikácie, a to je možnosť zobrazenia aplikácie na rôznych zariadeniach od mobilu až po veľkoformátové televízie. Problém zobrazovania jednej aplikácie na rôzne veľkých zariadeniach sa rieši aplikovaním techniky Responsive Design. Web, ktorý aplikuje túto techniku, sa pomocou špeciálnych CSS pravidiel a jazyka JavaScript prispôsobuje veľkosti obrazovky tak, aby bol dôležitý obsah vždy zreteľne zobrazený a návštevník sa tak mohol bezproblémovo dostať k požadovaným informáciám. Pri aplikovaní princípov Responsive Design sa niektoré časti stránky skrývajú na menších zariadeniach, alebo sú prístupné iba na vyžiadanie.

Jednou z ďalších nevýhodou webových aplikácií je nutnosť komunikovať so serverom, a teda aj nutnosť pripojenia na internet. Výpadok alebo zlá kvalita pripojenia môže obmedziť alebo dokonca znemožniť danú aplikáciu používať. Webovým aplikáciám chýba oproti natívnym aplikáciám napríklad aj možnosť priamej komunikácie so súborovým systémom alebo prístup ku hardvéru zariadenia ako napr. Bluetooth alebo USB pripojenie.

3.1. Progresívne webové aplikácie

Na odstránenie spomenutých negatív vznikli tzv. progresívne webové aplikácie (PWA). PWA kombinujú flexibilitu a dostupnosť webu s funkcionalitou natívnych aplikácií. Postavenie PWA medzi natívnymi a webovými aplikáciami je zobrazene na Obr. 2. Aby sme webovú aplikáciu mohli nazvať PWA, musí spĺňať nasledujúce kritéria:

- Načítanie cez HTTPS. Dnes je už načítanie webových aplikácií cez HTTPS bežnou praxou. Zabezpečené pripojenie zvyšuje dôveryhodnosť a umožňuje používateľovi využívať napríklad platby cez internet. HTTPS je vyžadované aj na sprístupnenie funkcionality, ktorá poskytuje citlivé dáta, ako napríklad polohu, kde sa zariadenie nachádza.
- Použitie skriptu service worker. Service worker je skript, ktorý dokáže kontrolovať sieťovú komunikáciu so server, ukladať jednotlivé volania do lokálnej pamäte a v prípade výpadku internetu tieto dáta poskytnúť.
- Existencia manifest súboru. V tomto súbore sú uložené základné informácie o aplikácií. Vo formáte JSON sa tu nastavuje názov aplikácie, farebná schéma, ikony a iné parametre, ktoré premenia webovú aplikáciu na aplikáciu, ktorá vyzerá a poskytuje funkcionalitu natívnych aplikácií. [3]



Obr. 2: Porovnanie dostupnosti a funkcionality natívnych, webových a PWA aplikácií [4]

Medzi ďalšie výhody PWA patrí možnosť inštalácie aplikácie na zariadenie, a umožnenie používať aplikáciu priamo, bez nutnosti pristupovať k aplikácií cez webový prehliadač. PWA prinášajú pocit používania natívnych aplikácií, s množstvom výhod ako napr. používanie v režime bez pripojenia na internet, upozornenia pomocou push notifikácií alebo zisťovanie polohy.

3.2. Prehľad technológií na tvorbu webových stránok a aplikácií

Existuje veľké množstvo programovacích jazykov, z ktorých si môže vývojár vybrať ten správny pre svoju aplikáciu. Keď však hovoríme o webových aplikáciách, vždy prídeme do kontaktu s HTML, CSS a programovacím jazykom JavaScript. HTML je jadrom každého webu a slúži na vytvorenie jeho obsahu v konkrétnej štruktúre a rozložení. Aby webové stránky neboli jednotvárne, pridávajú sa na stránky kaskádové štýly (CSS), ktoré definujú ako bude stránka vyzerať. Pomocou CSS sa nastavuje napr. farba pozadia, veľkosť a typ písma, rôzne animácie a mnoho iných vizuálnych vecí. Pre túto diplomovú prácu je však zo spomenutej trojice najpodstatnejší programovací jazyk JavaScript.

3.2.1. JavaScript

JavaScript je skriptovací, interpretovaný programovací jazyk. Aj keď je najznámejší ako jazyk pre webové stránky, svoje zastúpenie má aj mimo prostredia prehliadača a to napríklad ako prostredie Node.js, ktoré umožňuje spúšťať JavaScript kód aj na serveri, alebo Apache CouchDB databáza. JavaScript je jednovláknový, dynamický jazyk, ktorý podporuje objektovo-orientované, imperatívne a deklaratívne programovacie paradigmy. [5]

JavaScript bol vyvinutý Brendanom Eichom zo spoločnosti Netscape a pôvodne sa volal Mocha. Jeho meno bolo zmenené na JavaScript tesne pred uvedením tohto jazyka na verejnosť a to hlavne vďaka veľkej popularite jazyka Java v tej dobe. JavaScript avšak okrem podobnosti mena má s jazykom Java spoločné veľmi málo. [5]

JavaScript je interpretovaný jazyk, čo znamená, že na spustenie programu je potrebný priamo zdrojový kód a interpreter, ktorý tento kód vykonáva. Program v jazyku JavaScript sa teda nekompiluje do strojového kódu na to, aby bol spustiteľný.

JavaScript je taktiež dynamický programovací jazyk. Jedná sa o programovací jazyk, v ktorom je možné za behu programu vykonávať operácie ktoré sú inak vykonávané v čase kompilácie. Napríklad v jazyku JavaScript je možné zmeniť typ premennej alebo pridať nové vlastnosti alebo metódy k objektu, keď je program spustený. [6]

Dynamická povaha jazyka JavaScript je jednou z hlavných výhod, avšak zároveň je to aj jeho slabá stránka. JavaScript je taktiež slabo typový jazyk, čo znamená, že nemusíme definovať akého typu je premenná, a pri požití premennej v rôznom kontexte sa chová ako iný dátový typ.

```
const x = 2;
const y = "10";
const z = (x + y) * 5;

if (y == 10) {
    zavolajPrvuFunkciu();
}

if (y === 10) {
    zavolajDruhuFunkciu();
}
```

Obr. 3: Ukážka problémov vyplývajúcich zo slabej typovosti jazyka JavaScript

Na predchádzajúcom obrázku je demonštrované, aké problémy môže priniesť slabo typový programovací jazyk. Pri výpočte hodnoty premennej *z* môžeme uvažovať dvomi spôsobmi:

1. `z = (2 + 10)*5 = 60`
2. `z = ("2" + "10")*5 = "210" * 5 = 1050`

Oba prístupy sú správne, a kým s prvým spôsobom výpočtu sa môžeme stretnúť v jazyku Visual Basic, s druhým sa stretneme práve v jazyku JavaScript. Ďalším špecifickým znakom tohto jazyka je porovnávanie rovnosti hodnôt operátorom s tromi =. Na Obr. 3 vidíme dve podmienené volania funkcie *zavolajPrvuFunkciu* a *zavolajDruhuFunkciu.* Prvá funkcia sa zavolá, keďže pri vyhodnocovaní *if* podmienky sa premenná *y* berie ako číslo aj napriek tomu, že sme ju inicializovali na textovú hodnotu. Dochádza tu k automatickému pretypovaniu. V druhom prípade sa však funkcia nezavolá, pretože pri porovnávaní hodnôt pomocou logického operátora === sa porovnáva hodnota a zároveň aj typ premennej resp. typ porovnávanej hodnoty.

Postupom času začali vznikať rôzne projekty na rozšírenie funkcionality a eliminovanie slabých stránok jazyka JavaScript. Medzi hlavných zástupcov patrí TypeScript, CoffeScript alebo Flow. Ďalej sa v práci budem zaoberať hlavne jazykom TypeScript, ktorý je z trojice spomínaných projektov najpoužívanejší.

3.2.2. TypeScript

TypeScript je programovací jazyk vytvorený spoločnosťou Microsoft, s otvoreným zdrojovým kódom a širokou komunitou vývojárov. TypeScript je vytvorený ako nadmnožina jazyka JavaScript a finálny zdrojový kód je kompilovaný do jazyka JavaScript. Fakt, že TypeScript je nadmnožinou jazyka JavaScript znamená, že akýkoľvek platný JavaScript kód je spustiteľný aj v jazyku TypeScript. Toto prináša veľkú výhodu pri používaní existujúcich knižníc, bez nutnosti ich transformácie do nového jazyka. TypeScript prináša programátorovi rôzne užitočné vlastnosti, ktoré poznáme z iných programovacích jazykov, ako napríklad triedy a rozhrania, silnú typovosť, dedičnosť, generické typy alebo modifikátory prístupu k metódam triedy. [7]

Na nasledujúcom obrázku na ďalšej strane môžeme vidieť ukážku zdrojového kódu v jazyku TypeScript. Táto ukážka obsahuje definíciu rozhrania *Vehicle*, ktoré potom implementuje trieda *Car.* Modifikátori viditeľnosti je možné vidieť pri definovaní parametrov konštruktora. Zápisom *public*

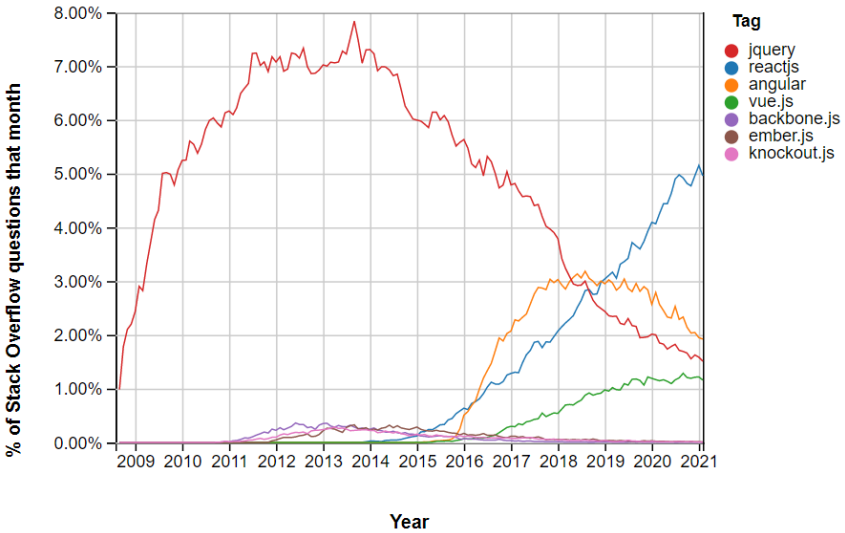
power: number definujeme atribút triedy a nastavíme na hodnotu, ktorú vložíme do konštruktora pri vytváraní inštancie.

```
interface Vehicle {
    name: string;
    power: number;
    maxSpeed: number;
    // ...
}

class Car implements Vehicle {
    name: string;
    constructor(public power: number, public maxSpeed: number) {
        this.name = `I'm a Car`;
    }

    // ... other methods
}

const myCar: Vehicle = new Car(130, 250);
```

Obr. 4: Ukážka zdrojového kódu v jazyku TypeScript

3.3. Knižnice a aplikačné rámce na vývoj webových aplikácií

Už dlhší čas sa vývoj webových aplikácií teší veľkej popularite. Pre zjednodušenie vývoja a uľahčenie riešenie stále dokola sa opakujúcich problémov vznikli rôzne knižnice a aplikačné rámce, ktoré nám pomáhajú pri tvorbe webových aplikácií. Vďaka nim sa za posledné desaťročie zmenil spôsob vývoja webových aplikácií, ale aj jednoduchých webových stránok. Zvýšila sa miera interakcie návštevníka s webom a výrazne vzrástla aj komplexnosť webových stránok. Na nasledujúcom obrázku je možné vidieť trend vo vývoji popularity knižníc používaných na vývoj webových aplikácií. Dáta sú získané z diskusného fóra Stack Overflow.



Obr. 5: Vývoji popularity webových technológií [8]

V súčasnosti sú najpoužívanejšie technológie Angular, ReactJS a Vue.js. Každá z týchto technológií má svoje klady a zápory. Líšia sa nielen syntaxou zdrojového kódu, ale aj množstvom funkcionality, ktoré poskytujú. Na nasledujúcich riadkoch budú popísané základne prvky jednotlivých technológií.

3.3.1. ReactJS

ReactJS (alebo aj React alebo React.js) je knižnica vytvorená spoločnosťou Facebook s otvoreným zdrojovým kódom, ktorá je určená na vývoj dynamických používateľských rozhraní webových aplikácií. Bola vytvorená pre interné použitie na stránkach Facebooku, a prvý krát bola nasadená v roku 2011 na zoznam noviniek, ktorý sa pravidelne obnovoval. V roku 2013 bola predstavená na konferencií JSConf a odvtedy si vybudovala pevne miesto medzi webovými technológiami. React umožňuje vývojárom vytvárať veľké webové aplikácie, ktoré môžu meniť údaje bez opätovného načítania stránky. [9]

Hlavný zámer tvorcov bolo vytvoriť knižnicu, ktorá efektívne zobrazuje veľké množstvo dát, ktoré sa menia v čase. Ako bolo spomenuté v predchádzajúcich kapitolách, obsah každej webovej stránky je tvorený pomocou HTML. Pri otvorení webovej stránky prehliadač z HTML kódu vytvorí takzvaný DOM (Document Object Model), čo je objektovo orientovaná reprezentácia HTML dokumentu. DOM umožňuje prístup a manipuláciu s obsahom dokumentu, jeho štruktúrou alebo vzhľadom. Veľký problém dynamických aplikácií je priama manipulácia s DOM, ktorá je náročná na zdroje. Problém nemusí vzniknúť pri menších aplikáciách, avšak neefektivita manipulácie s DOM sa prejaví ak na stránke je veľké množstvo prvkov (napríklad dlhé zoznamy alebo tabuľky s tisíckami záznamov). Tu prichádza najväčšia prednosť knižnice React a to je Virtual DOM.

Myšlienka Virtual DOM je vytvoriť kópiu skutočného DOM, ktorá existuje iba v pamäti prehliadača. V prípade zmeny dát je vygenerovaná nová reprezentácia Virtual DOM, a následné je porovnaná so súčasnou reprezentáciou. Algoritmus najprv zistí, ktoré uzly v celej hierarchii Virtual DOM boli zmenené, a až následné knižnica pristupuje k manipulácií so skutočným DOM. Tento proces je základom pre vysokú efektivitu a rýchlosť aplikácií vytvorených pomocou knižnice React. [9]

Čoraz viac webových aplikácií je vykresľovaných na strane klienta v prehliadači. HTML súbor, ktorý je poskytnutý serverom často neobsahuje okrem definovania štýlov a súborov s JavaScript kódom žiadny obsah. Celý obsah je tak načítaný a vykreslený na strane prehliadača. Toto spôsobuje, že vyhľadávacie roboty nevidia na stránke žiadny indexovateľný obsah. React rieši tento problém poskytnutím možnosťi vykreslenia aplikácie na serveri (SSR z anglického Server Side Rendering), čo pomáha pri SEO optimalizácií a taktiež tento prístup skracuje čas, ktorý je potrebný na to, aby sa návštevníkovi stránky zobrazil jej obsah.

Celá React aplikácia sa skladá z komponentov. Tieto komponenty môžu byť použité v iných komponentoch, a tak vytvárajú stromovú štruktúru. Správnym návrhom komponentov sa dosahuje vysoká miera znovupoužiteľnosti komponentov v rôznych častiach aplikácie. V neposlednom rade mála veľkosť knižnice (43.3kB gzip) napomáha k rýchlosti aplikácie. Pre tieto dôvody je React jednou z najpoužívanejších knižníc na tvorbu používateľského rozhrania. Na nasledujúcom obrázku je vidieť jednoduchá React aplikácia, ktorá ráta počet stlačení tlačidla na stránke. Pomocou funkcie *useState* je komponente vytvorené lokálne počítadlo. Funkcia vracia samotnú hodnotu *count*, a funkciu na

modifikáciu tohto stavu. Ďalej nasleduje definovanie štruktúry HTML, zobrazenie stavu počítadla pomocou zápisu *{count}* a inkrementovanie hodnoty počítadla pri kliknutí na tlačidlo.

```
import React, { useState } from 'react';

function ClickExample() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Tlačidlo bolo stlačené {count} krát</p>
      <button onClick={() => setCount(count + 1)}>Zvýš hodnotu počitadla</button>
    </div>
  );
}
```

Obr. 6: Ukážka komponenty v ReactJS

Šablóna je definovaná pomocou jazyka JSX. Jedná sa o rozšírenie jazyka JavaScript, ktoré pripomína jazyk HTML. Oproti HTML však umožňuje zobrazovať hodnoty premenných, dynamicky nastavovať hodnoty atribútov a používať podmienky a cykly. Tento jazyk podporuje všetky HTML elementy a ich atribúty. Na zobrazenie komponentu v šablóne sa používa zápis *<MojKomponent />*. [9]

3.3.2. Angular

Angular je webový aplikačný rámec s otvoreným zdrojovým kódom. Bol vytvorený a je udržovaný tímom Angular Team v spoločnosti Google a komunitou dobrovoľníkov. Angular je plnohodnotný webový aplikačný rámec, ktorý implementuje *Model-View-ViewModel(MVVM)* architektúru podobnú architektúre *Model-View-Controller(MVC)*. [10] Jeho robustnosť je hlavná črta, ktorá ho odlišuje od knižnice React.

Prvá verzia bola napísaná v jazyku JavaScript a niesla názov AngularJS. Pri vývoji druhej verzie prešiel tento rámec zásadnými zmenami. Nakoniec si vývojový tím uvedomil, že vyvinuli tak zásadne odlišný aplikačný rámec, že by nebolo vhodné, aby zdieľali spoločný názov. Aby nedošlo k nedorozumeniam, vývojový tím sa rozhodol pri novom rámci odobrať z názvu „JS“ a teda používať iba pomenovanie Angular. Zaujímavosťou je, že pri zmene názvu nezačali počítať verziu knižnice od začiatku, ale zostali pri verzií 2. Preto prvé vydanie tejto knižnice nieslo označenie Angular 2. [11]

Angular je na rozdiel od svojho predchodcu napísaný výhradne v jazyku TypeScript. Vývojový tím sa zaviazal vydať novú *major* verziu každých 6 mesiacov, a preto sa Angular stále rýchlo vyvíja, čo môže spôsobovať problém so spätnou kompatibilitou.

Podobne ako v React aplikácií, aj v Angular aplikácií sú základným stavebným prvkom komponenty. Každá aplikácia má práve jeden koreňový komponent, a spolu s ostatnými komponentami vytvárajú stromovú štruktúru. Každý komponent je trieda anotovaná dekorátorom *@Component,* a k nej prislúchajúca HTML šablóna, ktorá môže byť definovaná priamo v tele dekorátora alebo v samostatnom HTML súbore.

Úloha komponentu je zobrazovať dáta a nemal by obsahovať logiku aplikácie. Pre samotnú aplikačnú logiku slúžia služby. Sú to triedy anotované dekorátorom *@Injectable* a spravidla ich názov je

zakončený slovom *Service,* napr. *ArticleService*. Výhodou oddelenia aplikačnej logiky od komponenty je výrazne jednoduchšie testovanie a zdieľanie kódu. Služba sa do komponentu vkladá pomocou techniky vkladania závislosti (Dependency Injection), čo je v objektovo orientovanom programovaní technika vkladania závislosti medzi jednotlivými komponentami v prípade, že nemáme kontrolu nad vytváraním inštancií jednotlivých komponent. Inštanciu služby tak poskytne aplikačný rámec, namiesto toho, aby ju vytvoril komponent.

Ďalšia užitočná funkcionalita sú takzvane rúry (pipes). Tento mechanizmus umožňuje transformovať hodnotu výrazu v HTML šablóne. Príklad rúry *date* je zobrazený na nasledujúcom obrázku. Veľkou výhodou rúr je, že môžeme požiť ľubovoľný počet rúr za sebou pri transformácií jednej hodnoty. Angular poskytuje sadu predpripravených rúr, ktoré môžeme použiť. Ak však neposkytujú žiadanú funkcionalitu, je možné si vytvoriť vlastné. Na nasledujúcom obrázku je ukážka Angular komponentu s definovaním šablóny v dekorátore. Na príklade je tiež možné vidieť vkladanie služby *HttpClient* cez konštruktor triedy.

```
@Component({
    selector: 'app',
    template: `
    <button (click)="increment()">Zvýš hodnotu počitadla</button>
    <p>{{counter}}</p>
    <p>{{new Date() | date:`short` }}</p>
`
})
export class App {
    public counter: number = 0;

    constructor(private http: HttpClient) { }

    increment(){
      this.counter += 1;
    }
}
```

Obr. 7: Ukážka komponenty v aplikačnom rámci Angular

Angular poskytuje vývojárom mnoho ďalších funkcií, od zabudovaného systému na správu navigácie medzi stránkami, služby na HTTP komunikáciu so serverom alebo prednastaveného prostredia na testovanie. Angular taktiež ponúka nástroj Angular CLI, pomocou ktorého môže vývojár jednoducho vytvárať komponenty a služby z príkazového riadku.

3.3.3. Vue.js

Vue.js je webový aplikačný rámec s otvoreným zdrojovým kódom. Podobne ako predchádzajúce technológie, aj Vue.js sa používa na vytváranie používateľských rozhraní. Svojou veľkosťou, rýchlosťou a filozofiou sa podobá na knižnicu ReactJS. Využíva model založený na vytváraní a následnom skladaní komponentov do stromových štruktúr. Na zabezpečenie vysokého výkonu aplikácie využíva Virtual DOM (podobne ako ReactJS) pri rozhodovaní, ktoré komponenty je nutné prekresliť.

Podobnosť s knižnicou ReactJS vyplýva z faktu, že Vue.js bol ňou pri vývoji do veľkej miery inšpirovaný. Medzi hlavné rozdiely patrí menšia veľkosť knižnice a tým aj rýchlejšie načítanie aplikácie. Ďalším rozdielom je forma, akou sú definované komponenty. Na nasledujúcom obrázku môžeme vidieť príklad komponenty vo Vue.js, ktorý je do veľkej miery odlišný od React komponentu.

```
Vue.component('button-counter', {
    data: function () {
      return {
        count: 0
      }
    },
    template: `
    <button v-on:click="count++">Klikni na mna</button>
    <p>{{ count }}</p>
`
})
```

Obr. 8: Ukážka komponenty vo Vue.js

3.4. Source Maps

Zdrojový kód webových aplikácií počas zostavovania produkčného vydania prechádza procesom redukcie veľkosti (po anglicky minification). Jedná sa o proces transformácie súboru takým spôsobom, že zatiaľ čo funkčnosť výstupného súboru zostáva nezmenená, veľkosť súboru je radikálne menšia. V kontexte jazyka JavaScript a webových aplikácií je táto technika obzvlášť užitočná, pretože pomáha znížiť čas potrebný na načítanie stránky. Výsledný kód je nečitateľný voľným okom. Počas tohto procesu sa premenujú lokálne premenné a funkcie, odstránia sa všetky medzery, komentáre a oddeľovače blokov. [12]

Zo zdrojového kódu, ktorý prešiel procesom redukcie veľkosti, je veľmi náročné zistiť kontext vzniknutej chyby, keďže názvy funkcií a premenných sa nezhodujú s pôvodným zdrojovým kódom. Na riešenie tohto problému existujú súbory, ktoré obsahujú mapovanie medzi zredukovaným a originálnym zdrojovým kódom, tzv. Source Map.

Na nasledujúcom obrázku je vidieť štruktúra súboru Source Map.

```
{
    version: 3,
    file: "out.js",
    sources: ["foo.js", "bar.js"],
    names: ["src", "maps", "are", "fun"],
    mappings: "AAgBC,SAAQ,CAAEA"
}
```

Obr. 9: Štruktúra Source Map súboru [13]

Tento súbor obsahuje nasledujúce informácie:

- číslo verzie Source Map súboru
- názov súboru s redukovaným zdrojovým kódom

- názvy všetkých súborov s pôvodným zdrojovým kódom
- názvy všetkých premenných a metód, ktoré sa vyskytujú v celom zdrojovom kóde.
- samotné mapovanie pôvodných a zredukovaných názvov premenných a metód. Toto mapovanie je zapísané pomocou Base64 VLQ (Variable Length Quantity), čo rapídne redukuje veľkosť Source Map súboru. [13]

3.5. Zhrnutie

Ako je vidieť, svet webových technológií je veľmi pestrý a rýchlo sa vyvíjajúci. Pri tomto procese neustálych zmien je často ľahké zaviesť do aplikácie chybu, ktorá môže používateľovi spôsobovať problémy. Ak takýto problém nastane, je veľmi dôležité vytvoriť záznam o vzniknutej chybe a informovať vývojový tím. V nasledujúcej kapitole predstavím pár nástrojov, ktoré je možné použiť na monitorovanie webových aplikácií.

4. Prehľad existujúcich nástrojov na zaznamenávanie udalostí

Odchytávanie, ukladanie a zobrazovanie chybových hlásení nie je vo svete informačných technológií žiadna novinka. Preto v tejto kapitole predstavím existujúce riešenia so zameraním sa na webové aplikácie. Pri skúmaní týchto technológií sa zameriavam na nasledujúce požiadavky:

- Schopnosť monitorovať webové aplikácie vytvorené pomocou technológií React a Angular
- Schopnosť zobraziť kontext aplikácie v akom chyba vznikla
- Cena za technológiu
- Plný prístup k uloženým dátam
- Prepojenie s GitLab správcom incidentov
- Jednoduchosť inštalácie a používania

4.1. Sentry

Sentry je monitorovacia platforma, ktorá pomáha monitorovať aplikácie v reálnom čase a poskytuje užitočné informácie pri diagnostikovaní a oprave chýb. Podporuje monitorovanie aplikácií napísaných v rôznych programovacích jazykoch, ako napríklad Python, PHP, Java, JavaScript a mnoho ďalších. [14] Sentry je možné využívať ako službu SaaS (z anglického Software as a service), a ponúka rôznu funkcionalitu na základe predplateného programu. Služba je ponúkaná v troch platených balíkoch a jednom zadarmo. V nasledujúcej tabuľke je porovnanie jednotlivých balíkov.

Balík	**Počet záznamov za mesiac**	**Dĺžka uchovania záznamov**	**Počet používateľov**	**Integrácia s GitLab**	**Cena za mesiac**
Developer	5 000	30 dní	1	Nie	0 $
Team	50 000	90 dní	Neobmedzene	Áno	26 $
Business	50 000	90 dní	Neobmedzene	Áno	80 $
Enterprise	dohodou	dohodou	Neobmedzene	Áno	Dohodou

Tabuľka 1: Porovnanie balíkov od Sentry [15]

Sentry ponúka taktiež možnosť nainštalovať si jednoduchú verziu na vlastný server a mať tak výhradný prístup k zozbieraným dátam a konfigurácií. Túto verziu je možne nainštalovať a nakonfigurovať pomocou nástroja Docker a Docker Compose, neposkytuje však úplnú funkcionalitu, ktorú poskytuje verzia SaaS. [16]

Sentry má dobrú podporu monitorovania webových technológií a dokáže zaznamenávať udalosti z aplikácií napísaných pomocou knižníc React, Angular, Vue.js a mnohých ďalších. Veľmi dôležitá je aj podpora Source Maps, keďže zdrojový kód webových aplikácií je často minifikovaný a kontext, v ktorom chyba nastala sa stáva ťažko čitateľný.

Inštalácia a prvotné spustenie je jednoduché a dobre popísané v dokumentácií. Na nasledujúcom obrázku je postup inštalácie a nastavenie pre React webovú aplikáciu v troch krokoch.

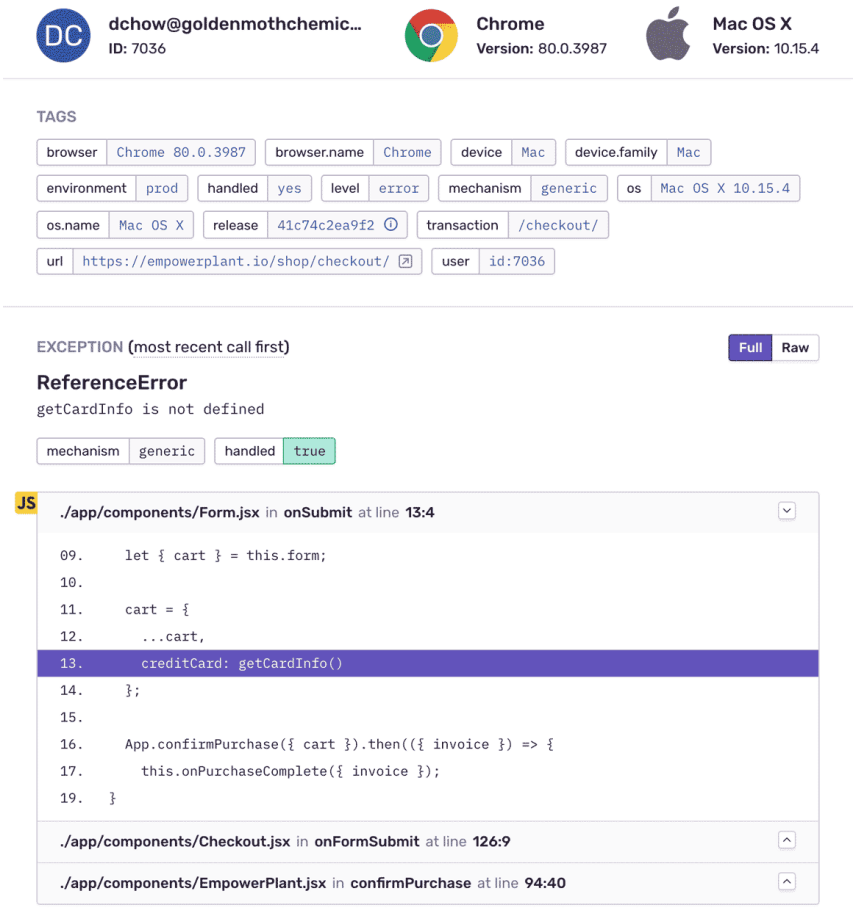
```
// 1. inštalácia knižnice
npm install --save @sentry/react

// 2. nastavenie knižnice v koreňovom komponente
import * as Sentry from "@sentry/react";
Sentry.init({ dsn: "https://examplePublicKey@o0.ingest.sentry.io/0" });

// 3. použitie komponentu na zachytávanie chýb
<Sentry.ErrorBoundary fallback={"An error has occurred"}>
  <MainComponent />
</Sentry.ErrorBoundary>
```

Obr. 10: Nastavenie Sentry pre React aplikáciu

Sentry zachytáva rôzne informácie o tom, v akom prostredí chyba vznikla. Medzi najdôležitejšie patrí:

- typ a verzia prehliadača
- platforma, z ktorej bola aplikácia používaná
- naformátovaný zdrojový kód aplikácie v mieste, kde došlo k chybe.

Na nasledujúcom obrázku je možné vidieť detail chybového záznamu.



Obr. 11: Detail chybového záznamu v Sentry

Samozrejmosťou sú rôzne grafy, ktoré pomáhajú pri získavaní rýchleho prehľadu o situácií. Ďalej Sentry ponúka možnosť rozsiahleho filtrovania záznamov, čo je užitočné pri odhaľovaní širšieho kontextu chyby.

Tento nástroj je veľmi obľúbený a rozšírený, používajú ho aj veľké spoločnosti ako napr. Disney alebo Microsoft. Je dobre odladený a vývojárskemu tímu poskytuje podporu v prípade, ak by došlo k nejakej neočakávanej chybe, alebo by vývojový tím potreboval pomôcť s konfiguráciou. Jeho hlavnou nevýhodou je, že v bezplatnej verzií poskytuje iba minimum funkcionality a umožňuje uložiť iba obmedzený počet chybových záznamov.

4.2. Rollbar

Ďalším nástrojom na monitorovanie aplikácií je Rollbar. Je to cloudové riešenie na sledovanie a monitorovanie chýb pre organizácie všetkých veľkostí. Rollbar podporuje monitorovanie aplikácií v rôznych programovacích jazykoch a technológiách, ako napríklad JavaScript, Python, .NET, Drupal, WordPress a mnohé iné. Rollbar poskytuje automatické zoskupovanie chýb na základe ich typu. Taktiež poskytuje možnosť prispôsobiť pravidlá, podľa ktorých sa záznamy zoskupujú do skupín. Ďalej je možné nastaviť upozornenia podľa závažnosti chyby. Riešenie tiež ponúka možnosti riadenia stavu problému, a umožňuje tak označiť problémy ako aktívne, vyriešené alebo stlmené. [17]

Rollbar umožňuje používateľom sledovať jednotlivé nasadenia aplikácie, poskytuje prehľadné nástenky vo forme grafov a umožňuje udržiavať históriu nasadení. K dispozícii je na základe mesačného alebo ročného predplatného, ktoré zahŕňa podporu prostredníctvom e-mailu. V nasledujúcej tabuľke sa nachádza porovnanie jednotlivých balíkov.

Balík	**Počet záznamov za mesiac**	**Dĺžka uchovania záznamov**	**Počet používateľov**	**Integrácia s GitLab**	**Cena za mesiac**
Free	5 000	30 dní	Neobmedzene	Áno	0$
Essentials	Od 6 000 do 4 000 000	180 dní	Neobmedzene	Áno	Od 1$ do 499$
Advanced	Od 100 000 do 4 000 000	180 dní	Neobmedzene	Áno	Od 83$ do 833$
Enterprise	dohodou	dohodou	Neobmedzene	Áno	Dohodou

Tabuľka 2: Porovnanie balíkov platformy Rollbar [18]

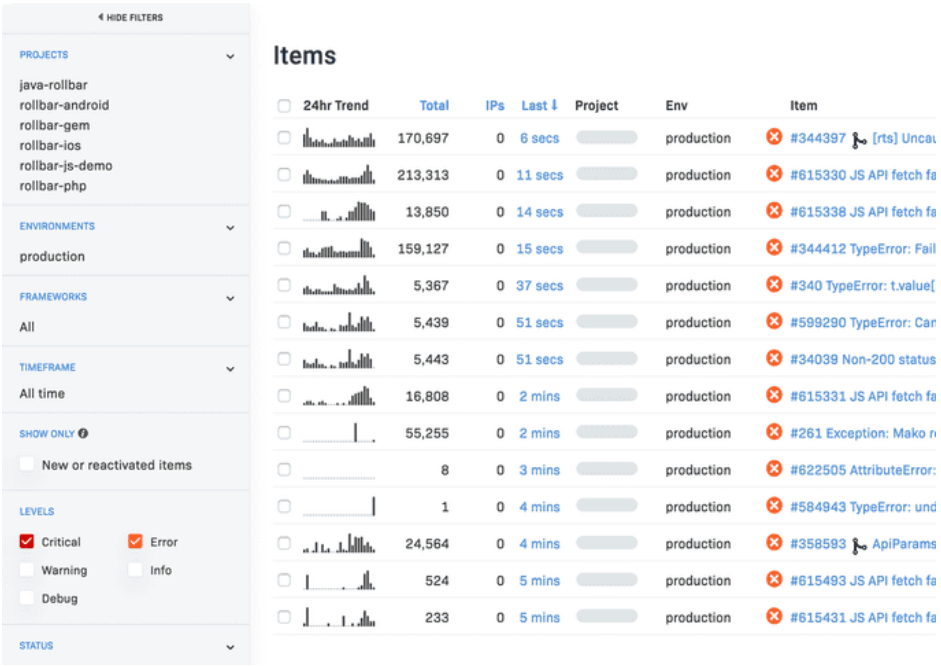
Rollbar podporuje najpoužívanejšie webové technológie vrátane knižníc React a Angular. Dôležitá je aj podpora Source Maps v základnom balíku. Výhodou je integrácia s platformou Slack, kde môžu vývojári dostávať upozornenia do komunikačného vlákna. V bezplatnom balíku je aj možnosť integrácie s nástrojmi na správu incidentov ako napr. GitLab alebo JIRA.

Inštaláciu a konfiguráciu tohto nástroja je vidieť na Obr. 12 na ďalšej strane. Rollbar neobsahuje žiadne špeciálne komponenty pre jednotlivé knižnice a aplikačné rámce, a teda zaznamenávanie chýb neprebieha automaticky. Použitie tejto knižnice je rovnaké naprieč všetkými webovými technológiami využívajúcimi jazyk JavaScript. Nevýhodou tohto riešenia je, že sa o zaznamenanie chyby musí postarať vývojár vo vlastnom kóde.

```
// 1. inštalácia knižnice
npm install --save @sentry/react

// 2. inicializ8cia knižnice
import Rollbar from "rollbar";

const rollbar = new Rollbar({
  accessToken: 'POST_CLIENT_ITEM_TOKEN',
  captureUncaught: true,
  captureUnhandledRejections: true,
});

// 3. manuálne logovanie chyby
rollbar.error('react test error');
```

Obr. 12: Inštalácia a konfigurácia nástroja Rollbar

Na nasledujúcom obrázku je ukážka používateľského rozhrania, kde je zobrazený zoznam záznamov zoskupených podľa chyby. Rozhranie poskytuje všetky potrebné informácie, nie je však tak prehľadné ako pri nástroji Sentry, a môže pôsobiť zastaralým dojmom.



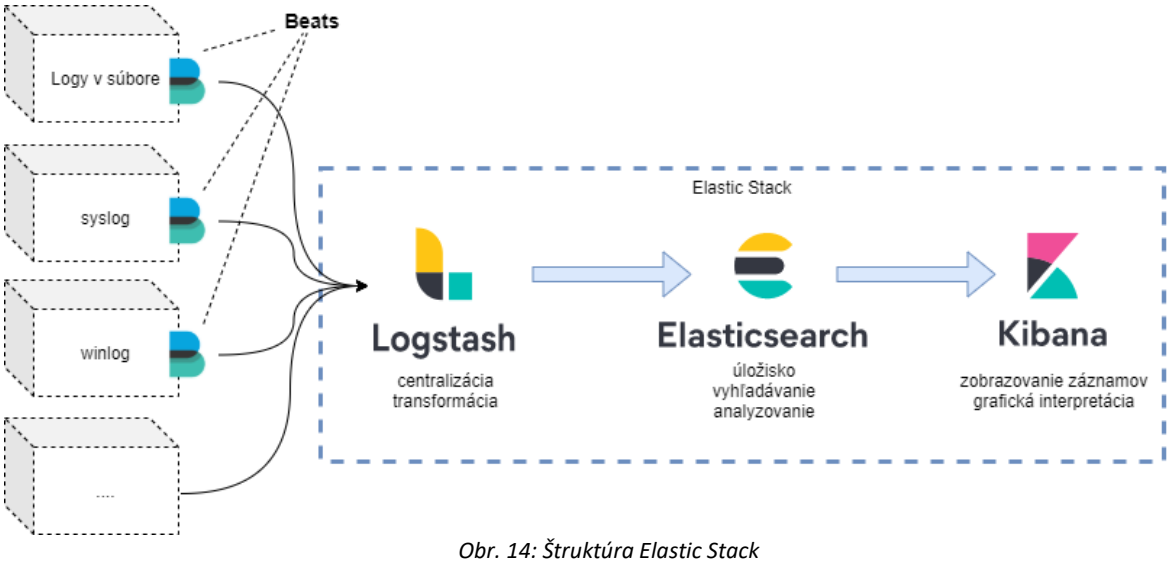
Obr. 13: Zobrazenie zoznamu zachytených chybových hlásení v nástroji Rollbar [19]

4.3. Elastic Stack

„ELK Stack“ je skratka pre tri projekty s otvoreným zdrojovým kódom: Elasticsearch, Logstash a Kibana. Elasticsearch je vyhľadávací a analytický nástroj. Logstash je server na spracovanie údajov, ktorý zhromažďuje údaje z viacerých zdrojov súčasne, transformuje ich a potom ich posiela na uloženie do iného nástroja, napr. do Elasticsearch. Kibana umožňuje používateľom vizualizovať údaje

z Elasticsearch pomocou grafov. Elastic Stack je evolúciou pomenovania ELK Stack, pričom základné nástroje zostali nezmenené.

Jedno z rozšírených využití platformy Elastic Stack je práve zbieranie a uchovávanie chybových hlásení. Na zaznamenávanie a prenos chybových hlásení z koncových zariadení sa používajú tzv. Beats. Sú to jednoduché programy nainštalované na koncových zariadeniach, ktorých jedinou úlohou je odosielať údaje do nástroja Logstash. Na nasledujúcom obrázku je znázornená štruktúra platformy Elastic Stack. V ďalších podkapitolách sú popísané jednotlivé prvky.



Obr. 14: Štruktúra Elastic Stack

4.3.1. Elasticsearch

Elasticsearch je distribuovaný vyhľadávací a analytický nástroj s otvoreným zdrojovým kódom. Je to škálovateľné riešenie pre uloženie JSON dokumentov bez fixnej schémy a zároveň poskytuje fulltextové a parametrické vyhľadávanie. Je postavaný na fulltextovom vyhľadávacom nástroji Apache Lucene. [20]

Elasticsearch na rozdiel od Apache Lucene prináša model horizontálneho škálovania využitím techniky replikácie. Hlavné využitie v dnešnej dobe nachádza pri rôznych vyhľadávacích nástrojoch ale aj ako úložisko pre log záznamy. Na komunikáciu využíva REST API a vyhľadávací dopyt, ako aj odpoveď zo servera, je vo formáte JSON. [20]

Elasticsearch terminológia

Na začiatok je žiadúce si definovať terminológiu, ktorá je neoddeliteľnou súčasťou Elasticsearch ekosystému a tieto termíny budú ďalej použité v diplomovej práci. Definície jednotlivých termínov sú čerpané z knihy *Elasticsearch: The Definitive Guide.* [20]

Node (uzol)– jedna inštancia Elasticsearch (môže ale nemusí byť samostatný server), ktorá sa podieľa na indexovaní a vyhľadávaní dokumentov.

Cluster (klaster) – zoskupenie jedného alebo viacerých uzlov. Zabezpečuje riadenie indexovania dokumentov ako aj vyhľadávanie naprieč všetkými uzlami.

Document (dokument) – základná dátová jednotka uložená v Elasticsearch vo forme JSON objektu. V kontexte ukladania log záznamov sa jedná o jeden záznam o chybe alebo udalosti. Každý dokument má unikátny identifikátor a patrí do práve jedného indexu.

Index – logická množina jedného alebo viacerých dokumentov s rovnakými alebo veľmi podobnými charakteristikami.

Shard – pojem shard sa používa na označenie častí indexu. Keďže jednotlivé uzly môžu byť nasadené na samostatných serveroch, jeden index môže byť rozdelený na menšie časti, ktoré sú umiestnené v jednotlivých uzloch v rámci klastra. Shard slúži na distribúciu dát tak, aby boli lepšie využité prostriedky všetkých uzlov. Týmto spôsobom je zabezpečené rýchle indexovanie vďaka paralelným procesov, a taktiež rýchle vyhľadávanie, čo je kľúčová priorita pre Elasticsearch.

Replica – predstavuje ochranný mechanizmus pred stratou dát v prípade zlyhania uzla alebo straty časti indexu (shard). Repliky sú v podstate kópie všetkých častí indexu, ktoré sú spravidla uložené na inom uzle pre prípad zlyhania pôvodného uzla.

Relačná databáza	**Elasticsearch**
Tabuľka	Index
Záznam (riadok) v tabuľke	Document
Stĺpce tabuľky	Fields
Schéma	Mapping
SQL	Query DSL
SELECT * FROM table WHERE id = 1	GET http://localhost:9200/index/_doc/1
UPDATE table SET ...	PUT http://localhost:9200/index/_doc

Tabuľka 3: Porovnanie pojmov relačnej databázy a Elasticsearch

4.3.2. Logstash

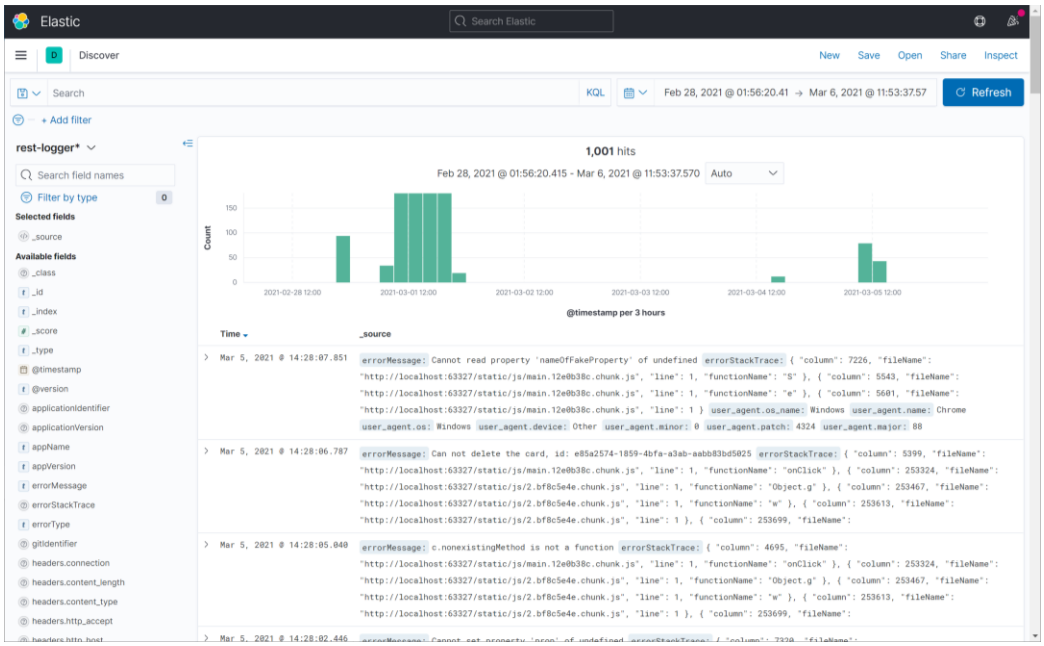
Logstash je program, ktorý prijíma, transformuje a sprostredkúva dáta bez ohľadu na formát alebo zložitosť. Odvodzuje štruktúru od neštruktúrovaných údajov pomocou funkcie grok, dešifrujte geografické súradnice z IP adries, anonymizujte alebo odstraňuje citlivé dáta a uľahčuje celkové spracovanie rôznych dát. [21]

Na prenos dát do nástroja Logstash z miesta, kde vznikajú, sa využívajú malé programy, tzv. Beats. Tieto programy dokážu zbierať dáta z rôznych zdrojov. FileBeat zbiera záznamy z textových log súborov, WinlogBeat spracováva záznamy o udalostiach priamo z OS Windows alebo MetricBeat pravidelne zbiera metriky z operačných alebo databázových systémov. [22] Po spracovaní a transformácií dát na základe pravidiel nastavených v konfiguračnom súbore, sa dáta posielajú spravidla do Elasticsearch, kde sa uložia a pripravia na ďalšiu analýzu.

4.3.3. Kibana

Kibana je webová aplikácia s otvoreným zdrojovým kódom, ktorá je umiestnená na vrchole platformy Elastic Stack. Poskytuje používateľské rozhranie na vyhľadávanie a vizualizáciu dát uložených v

Elasticsearch. Kibana je aj nástroj na tvorbu grafov pre Elastic Stack, a slúži aj ako používateľské rozhranie pre monitorovanie, správu a zabezpečenie Elastic Stack klastra. [23] Na Obr. 15 vidíme webové rozhranie nástroja Kibana pri prezeraní uložených dát.



Obr. 15: Zobrazovanie záznamov v nástroji Kibana

4.4. Zhrnutie

Každý zo spomenutých nástrojov na zaznamenávanie a manipuláciu s log záznamami má svoje klady aj zápory. Jednoduchá inštalácia a nastavenie nástroja Sentry a Rollbar je vykúpená obmedzenou funkcionalitou v bezplatnej verzii. Jedna z požiadaviek na nástroj je prepojenie so systémom na správu incidentov. Túto požiadavku platforma Sentry nedokáže splniť v bezplatnej verzii. Bezplatné verzie oboch nástrojov Sentry a Rollbar navyše ponúkajú iba obmedzené množstvo zaznamenaných chybových hlásení za mesiac.

Elastic Stack na rozdiel od nástrojov Sentry a Rollbar je riešenie, ktoré beží na vlastnom serveri. Výhodou tohto riešenia je možnosť plnej kontroly nad konfiguráciou a taktiež priamy prístup k uloženým dátam. Na druhej strane toto riešenie prináša potrebu prvotnej konfigurácie platformy a následné spravovanie celého systému počas celej doby používania.

Elastic Stack poskytuje širokú infraštruktúru na zbieranie a ukladanie záznamov o udalostiach z rôznych zariadení. V kombinácií s priamym prístupom k dátam to robí z tejto platformy ideálny základ pre implementovanie vlastného, na mieru vytvoreného nástroja. V ďalšej kapitole je popísaný návrh aplikácie využívajúcej platformu Elastic Stack.

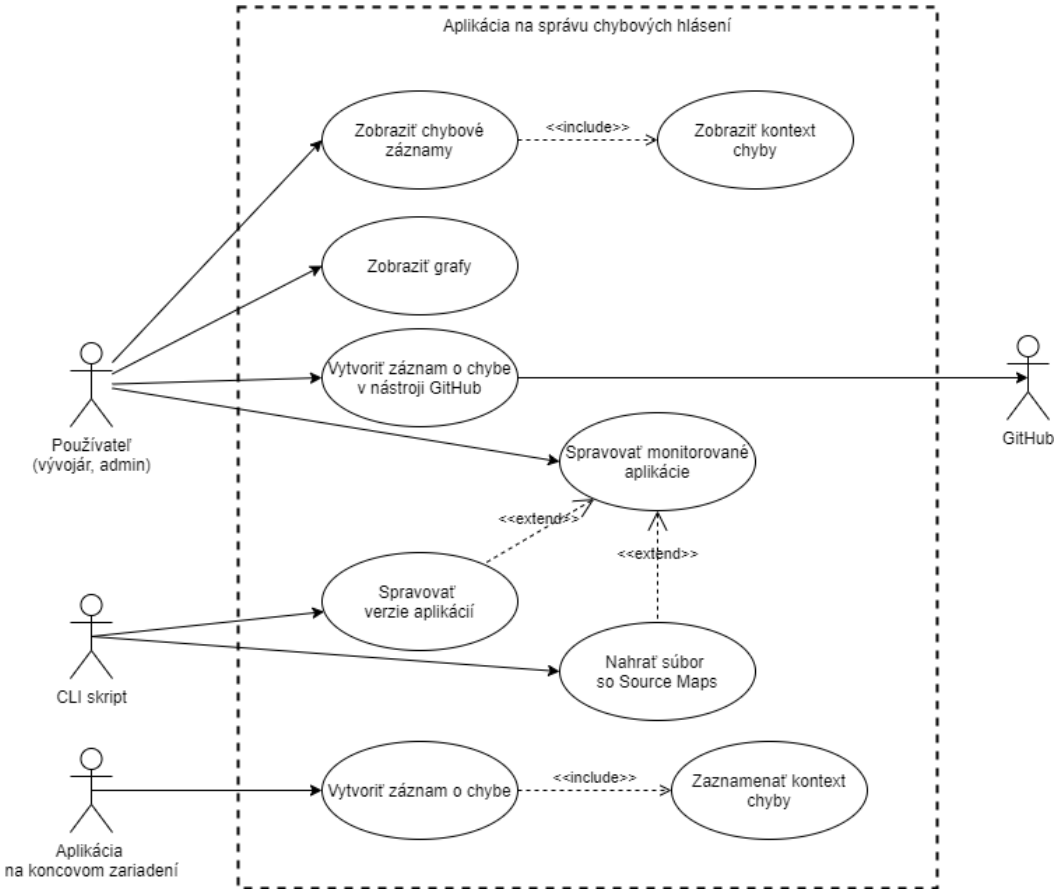
5. Návrh a implementácia aplikácie

Táto časť diplomovej práce sa venuje návrhu aplikácie. Postupne je predstavená celková architektúra aplikácie a následne sú popísané jednotlivé časti detailnejšie.

5.1. Požiadavky na aplikáciu

Po preskúmaní existujúcich riešení a rozhodnutí, že optimálnym riešením spravovania chybových hlásení bude vytvorenie vlastnej aplikácie, nastala fáza návrhu. Na to, aby navrhnutá aplikácia vhodne riešila daný problém, je v prvom rade nutné definovať, čo od aplikácie očakávame. Jedná sa o proces definovania požiadaviek. Existujú rôzne spôsoby, akými je možné požiadavky zapísať. Príkladom môžu byť číselné zoznamy, alebo tak ako je to v tomto prípade, diagramy UML.

Unified Modeling Language (UML) je štandard vizuálneho jazyka, ktorý slúži na modelovanie procesov, analýzu, návrh a implementovanie softvérových systémov. Jedná sa o spoločný jazyk pre analytikov, softvérových architektov a vývojárov. Slúži na popísanie, špecifikovanie a dokumentovanie existujúcich alebo nových procesov, štruktúr alebo správania sa softvérových systémov. [24] Na modelovania funkcionality, ktorú by mala aplikácia poskytovať, je vhodný diagram prípadov použitia (use case diagram), ktorý je možné vidieť na nasledujúcom obrázku

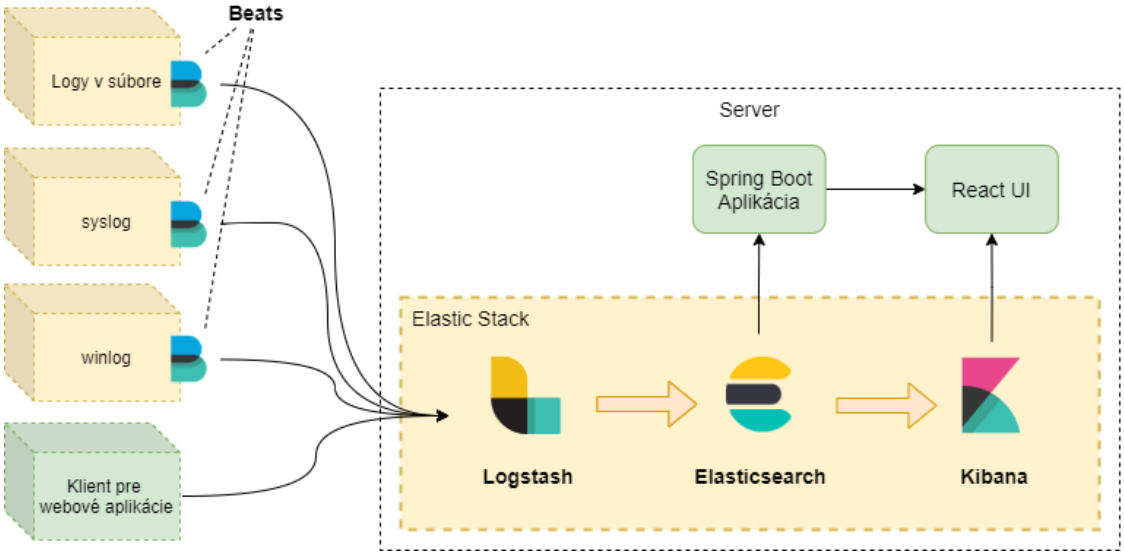


Obr. 16: Use case diagram na modelovanie funkcionality systému

Hlavným používateľom aplikácie na monitorovanie webových aplikácií je persóna vývojára alebo správcu systému. Takýto používateľ očakáva od systému hlavne prehľad monitorovaných aplikácií a zariadení. Pri vzniku problému je dôležité poskytnúť čo najviac relevantných informácií o chybe, ktoré prispejú k rýchlej oprave. Prepojenie jednotlivých chýb s nástrojom na správu incidentov umožní jednoduchšiu integráciu s už existujúcimi firemnými nástrojmi a procesmi. Pri vydaní novej verzie monitorovanej aplikácie prichádza na scénu ďalší aktér. Je ním skript, ktorý oznamuje systému existenciu novej verzie a nahráva súbory *Source Maps* na server. Nástroj tak môže novú verziu bez problémov monitorovať a zobrazovať správny zdrojový kód v detaile záznamu. Posledným účastníkom je monitorovaná aplikácia, v našom prípade sa jedná hlavne o webovú aplikáciu. Táto aplikácia musí byť v prípade vzniku chyby schopná chybu zachytiť a odoslať ju do centrálnej aplikácie na spracovanie a následné uloženie. Pre lepšie pochopenie problému by mal nástroj zaznamenať širší kontext chyby, a to vrátane množiny činností, ktoré vykonal používateľ a ktoré viedli ku vzniku chyby.

5.2. Architektúra aplikácie

Celá aplikácia je rozdelená na logické časti na základe toho, akú funkciu plnia v celom systéme. Server predstavuje miesto centralizovaného spracovania a ukladania hlásení, a aplikácie na monitorovaných zariadeniach predstavujú klientov. Pri skúmaní existujúcich riešení bola predstavená aj platforma Elastic Stack. Táto platforma poskytuje dobrý základ pre vlastnú implementáciu nástroja na ukladanie a spravovanie chybových záznamov z koncových zariadení.



Obr. 17: Architektúra celého riešenia

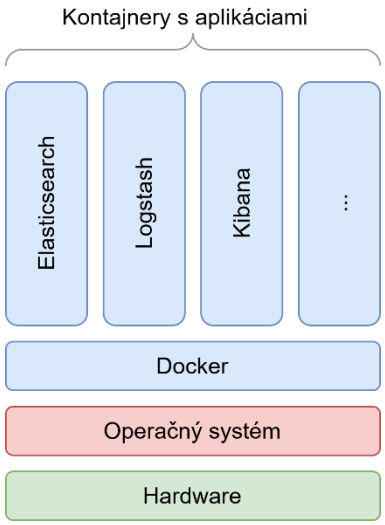
V nasledujúcej kapitole sa zaoberám inštaláciou a nastavením platformy Elastic Stack. Na tomto základe sú postavené ďalšie časti tohto systému. V ďalšej kapitole sa potom zoberám mechanizmom na zachytenie chybových a iných hlásení na koncovom zariadení a jeho odoslaním na server. Ďalšia kapitola je venovaná problematike serverovej aplikácie, ktorá slúži na správu a manipuláciu s uloženými dátami v databáze Elasticsearch. Poskytuje taktiež REST API pre používateľské rozhranie, ktorému je venovaná posledná kapitola.

5.3. Inštalácia a nastavenie Elastic Stack

Pre inštaláciu a nastavenie platformy Elastic Stack bol zvolený nástroj Docker.

5.3.1. Docker

Docker je otvorená platforma pre vývoj a prevádzkovanie aplikácií. Docker umožňuje oddeliť aplikáciu od infraštruktúry na ktorej beží, čo prispieva k rýchlemu nasadeniu nových vydaní aplikácie. Docker poskytuje možnosť zabaliť a spustiť aplikáciu v izolovanom prostredí, ktoré sa nazýva kontajner. [25] Princíp fungovania nástroja Docker je zobrazený na nasledujúcom obrázku.



Obr. 18: Princíp virtualizácie pomocou Docker kontajnerov [26]

Medzi hlavné výhody použitia nástroja Docker patrí nezávislosť aplikácie na operačnom systéme hostiteľského počítača. Aplikácia dokáže bežať na ľubovoľnom operačnom systéme, ktorý podporuje tento nástroj. Nezáleží teda na tom, či je aplikácia spustená na serveri s OS Linux, alebo na počítači vývojára s OS Windows. Ďalšia nesporná výhoda sa prejaví pri experimentovaní s rôznymi technológiami. Docker obsahuje množstvo predpripravených obrazov (image), ktoré môže vývojár použiť. Kedykoľvek môže zvolenú technológiu zmeniť, a bez zdĺhavej inštalácie a nastavovania tak môže zameniť jednu technológiu za druhú.

5.3.2. Konfigurácia platformy Elastic Stack

Docker vytvára prostredie, v ktorom beží aplikácia. Toto prostredie je vytvorené pri spustení obrazov, ktoré sa definujú v súbore *dockerfile.* Tento súbor obsahuje inštrukcie, ktoré určujú postupnosť krokov potrebných na to, aby pri spustení obrazu (spustený obraz sa nazýva kontajner) bolo prostredie nastavené podľa našich potrieb. Tieto kroky môžu zahŕňať napríklad inštaláciu nástrojov ako MySql databázy a jej nastavenie, kopírovanie konfiguračných súborov, vytváranie súborovej štruktúry a mnoho iného. Je to podobné, ako keď do čistej inštalácie Linuxu nainštalujeme potrebné nástroje a programy na to, aby bolo prostredie pripravené na beh našej aplikáciu.

Dôležitou vlastnosťou obrazov je, že môžu byť použitý ako základ v definícií iného obrazu. Toto umožňuje existenciu predpripravených obrazov technológií, ktoré sú často používané. Jedná sa o

rôzne databázy, webové servery, firewally a mnohé iné. Predpripravené obrazy sú často spravované priamo vývojármi daných technológií, ktorí garantujú správnosť konfigurácie danej technológie.

Elastic vytvoril Docker obrazy pre jednotlivé nástroje platformy Elastic Stack a teda ich inštalácia je veľmi jednoduchá. Pre každý nástroj je vytvorený vlastný *dockerfile.* Jeho obsah je vyobrazený na nasledujúcom obrázku. Ako je možné vidieť, *Dockerfile* pre jednotlivé nástroje je jednoduchý. Všetka konfigurácia nutná na spustenie nástroja je definovaná práve v obraze, ktorý je vytvorený spoločnosťou Elastic. Pri použití takto predpripraveného obrazu ostáva už len správne definovať správnu cestu k obrazu a verziu nástroja, ktorú chceme použiť.

```
// dockerfile pre elasticsearch
ARG ELK_VERSION
FROM docker.elastic.co/elasticsearch/elasticsearch:${ELK_VERSION}

// dockerfile pre logstash
ARG ELK_VERSION
FROM docker.elastic.co/logstash/logstash:${ELK_VERSION}

// dockerfile pre kibanu
ARG ELK_VERSION
FROM docker.elastic.co/kibana/kibana:${ELK_VERSION}
```

Obr. 19: Dockerfile pre Elastic Stack

Pre spustenie Docker obrazu je potrebné daný obraz najprv zostaviť pomocou príkazu `docker build` v príkazovom riadku v zložke, kde sa *dockerfile* nachádza. Po vykonaní tohto príkazu sa vytvorí spustiteľný obraz, ktorý sa uloží na disk. Prvotné zostavenie obrazu vyžaduje stiahnutie väčšieho množstva balíčkov a preto býva často časovo náročné. Nasledujúce zostavenia už nemusia sťahovať veľké množstvo balíkov, a tak každé ďalšie zostavenie je kratšie. Po zostavení je možné obraz spustiť pomocou príkazu `docker run`.

Keďže v tomto prípade pracujeme s viacerými obrazmi, na spustenie všetkých nástrojov by bolo potrebné vykonať príkaz `docker run` tri krát, individuálne pre každý obraz. Pre vytváranie a nastavenie komplexného prostredia pozostávajúceho z viacerých kontajnerov sa používa nástroj Docker Compose. Nastavenie pre tento nástroj je uložene v súbore *docker-compose.yml*, ktorý je zobrazený na Obr. 20 na ďalšej strane. Obsah tohto súboru je často dlhý, a z toho dôvodu je zobrazená konfiguráciu iba jednej služby, keďže konfigurácia ostatných sa vo veľkej miere opakuje.

V našom prípade konfiguračný súbor obsahuje dve sekcie: *services, networks.* V sekcii *services* sa definujú služby, z ktorých pozostáva celé prostredie. Každá služba musí mať unikátne meno v rámci služieb, aby mohla byť ďalej v konfigurácií adresovaná. Ďalej pre každú službu musí byť definovaný Docker obraz. Ten sa definuje pomocou parametru *build/dockerfile*. Ak však má tento súbor predvolený názov *dockerfile*, tak sa tento pramater môže vynechať. Ďalej sú definované argumenty, ktoré sa predávajú jednotlivým obrazom pri spustení. V tomto prípade sa posiela jednotná verzia pre všetky nástroje platformy Elastic Stack.

```
services:
  elasticsearch:
    ...

  logstash:
    build:
      context: logstash/
      args:
        ELK_VERSION: $ELK_VERSION
    volumes:
      - type: bind
        source: ./logstash/config/logstash.yml
        target: /usr/share/logstash/config/logstash.yml
        read_only: true
    ports:
      - "5044:5044"
      - "5045:5045"
    environment:
      LS_JAVA_OPTS: "-Xmx256m -Xms256m"
    networks:
      - elk
    depends_on:
      - elasticsearch

  kibana:
    ...

networks:
  elk:
    driver: bridge
```

Obr. 20: Nastavenie platformy Elastic Stack pomocou nástroja Docker Compose

Keďže kontajner je od okolitého prostredia izolovaný, je potrebné definovať rozhranie, pomocou ktorého komunikuje. Definovaním parametru *volumes* sa nastavuje prepojenie súborového systému medzi hostiteľským počítačom a kontajnerom. V tomto prípade je nastavené prepojenie tak, aby bol nástroju Logstash podsunutý konfiguračný súbor, ktorý je umiestnený na hostiteľskom počítači. Ďalší dôležitý parameter je *ports*. Pomocou tohto parametru sa definuje množina portov, ktoré sú prístupne pre ostatné služby a zároveň na hostiteľskom počítači. Porty sú definované textovým reťazcom vo formáte *"hostiteľský počítač:kontajner"*. Povoliť komunikáciu cez jednotlivé porty je veľmi dôležité pre to, aby nástroje platformy Elastic Stack mohli medzi sebou komunikovať.

Pomocou parametru *environment* sú nastavené premenné prostredia. V tomto prípade sa jedná o parametre pre Java Virtual Machine. Aby služby mohli medzi sebou komunikovať, sú napojene na jednu virtuálnu sieť pomocou parametru *network*. V komplikovanejších prípadoch môže konfiguračný súbor obsahovať viac virtuálnych sietí, čím sa za zabezpečí izolácia jednotlivých služieb. Posledným parametrom je parameter *depends_on*. Ako je z názvu zrejmé, pomocou tohto parametra určujeme závislosť na inej službe. Daná služba je tak spustená až po spustení služby, na ktorej je závislá.

Po nastavení konfiguračného súboru *docker-compose.yml* je možné celé prostredie spustiť. Vykonaním príkazu `docker-compose up` sa stiahnu všetky potrebné závislostí jednotlivých obrazov, tie sa následne zostavia a spustia. Podobne ako pri nástroji Docker, aj tu platí, že zostavovanie jednotlivých obrazov je nutné iba počas prvého spustenia. Pre urýchlenie nasledujúcich spustení sa použije zostavený obraz, ktorý je uložený v pamäti na disku.

Po nastavení prostredia, kde beží platforma Elastic Stack, je nutné nastaviť samotné nástroje tejto platformy. Nastavenie prebieha pomocou konfiguračných súborov, ktoré sú pripojené ku kontajnerom pomocou nastavenia v *docker-compose.yml.* Štruktúra všetkých konfiguračných súborov pre platformu Elastic Stack je zobrazená na nasledujúcom obrázku.

```
elk
 |-- elasticsearch
 |    |-- config/elasticsearch.yaml
 |    |-- Dockerfile
 |-- kibana
 |    |-- config/kibana.yaml
 |    |-- Dockerfile
 |-- logstash
 |    |-- config/logstash.yaml
 |    |-- pipeline/logstash.conf
 |    |-- Dockerfile
 |-- docker-compose.yaml
```

Obr. 21: Štruktúra konfiguračných súborov platformy Elastic Stack

Základná konfigurácia nástrojov platformy Elastic Stack je jednoduchá. Každý nástroj obsahuje konfiguračný súbor s predvolenými nastaveniami, a programátor tak musí zmeniť nastavenia iba určitých parametrov. Na nasledujúcom obrázku sú zobrazené konfiguračné súbory pre Elastic Stack.

```
# elasticsearch.yaml
cluster.name: "docker-cluster"
network.host: "0.0.0.0"
xpack.license.self_generated.type: basic

# logstash.yaml
http.host: "0.0.0.0"

# kibana.yaml
server.name: kibana
server.host: "0.0.0.0"
elasticsearch.hosts: [ "http://elasticsearch:9200" ]
```

Obr. 22: Konfigurácia nástrojov Elastic Stack

Konfiguračný súbor obsahuje nasledovné nastavenia:

- *cluster.name* – názov klastra, ku ktorému patrí inštancia Elasticsearch. Pomocou tohto parametru je možné jednotlivé inštancie Elasticsearch priraďovať do klastrov, ktoré sú od seba izolované. [27] V našom prípade beží iba jedna inštancia Elasticsearch, a teda existuje práve jeden klaster tvorený jednou inštanciou Elasticsearch.
- *network.host* – nastavenie sieťovej adresy, cez ktorú inštancia komunikuje. Nastavenie na hodnotu "0.0.0.0" znamená, že inštancia Elasticsearch automaticky priradí sieťovú adresu na základe sieťového rozhrania. [28]
- *http.host* – nastavenie sieťovej adresy, cez ktorú prebieha HTTP komunikácia. [28] Podobne ako pri parametri *network.host,* nastavenie na hodnotu "0.0.0.0" znamená automatické pridelenie adresy.
- *xpack.license.self_generated.type* – nastavenie základnej, bezplatnej licencie
- *elasticsearch.hosts* – URL inštancie Elasticsearch, ktorú Kibana používa na komunikáciu s Elasticsearch databázou.

Posledným chýbajúcim krokom je nastavenie nástroja Logstash. Logstash je server, ktorý spracováva a zhromažďuje dáta z viacerých zdrojov. Aby bol schopný prijať dáta z monitorovanej webovej aplikácie, je nutné ho správne nakonfigurovať.

Konfiguračný súbor je možné vidieť na Obr. 23 na nasledujúcej strane. Skladá sa z troch hlavných blokov, ktoré definujú zdroje dát, ich transformáciu a miesto, kde sú nakoniec odoslané.

Transport chybových záznamov z webovej aplikácie prebieha cez protokol HTTP. Pre nastavenie vstupného HTTP modulu je nutné nastaviť port, na ktorom daný modul komunikuje. Tento port musí byť následne povolený aj v konfiguračnom súbore *docker-compose.yml*. Ďalej je nastavený formát, v akom Logstash očakáva prijímané dáta. V tomto prípade sa jedná o dáta vo forme JSON.

Moderné prehliadače často z bezpečnostných dôvodov blokujú komunikáciu medzi rôznymi doménami. Keďže sieťová adresa serveru nástroja Logstash sa nemusí zhodovať s adresou, z ktorej je prístupná monitorovaná webová aplikácia, musí byť povolená komunikácia medzi rôznymi doménami nastavením hlavičky *Access-Control-Allow-Origin* na hodnotu *“*“*. Skrz toto nastavenie prehliadač vie, že nemá blokovať komunikáciu medzi Logstash serverom a webovou aplikáciou.

Transformácie dát sú nastavené v bloku *filter.* Nadefinované sú dve transformácie pomocou predpripravených transformačných filtrov:

- *date* – filter *date* slúži na analyzovanie a transformáciu dátumov rôznych formátov. V tomto prípade filter očakáva časovú známku vo formáte *UNIX_MS*, čo predstavuje počet milisekúnd od 1.1.1970. Následné je časová známka transformovaná do formátu ISO 8601 a uložená do atribútu *@timestamp.* Daný formát a názov atribútu je kompatibilný s predvoleným nastavením nástroja Kibana, a tak nie je potrebná dodatočná konfigurácia.
- *useragent* – tento filter analyzuje a transformuje *user agent* textový reťazec do štruktúrovanej formy, ktorá obsahuje informácie o prehliadači, operačnom systéme a zariadení.

Posledným blokom je *output* blok. V tomto bloku je nastavené miesto, kam sa prijaté a transformované dáta odosielajú. V tomto prípade je cieľovou destináciou nástroj Elasticsearch. Definovaná adresa inštancie obsahuje názov Elasticsearch služby z konfigurácie nástroja Docker Compose a port, na ktorom Elasticsearch počúva.

```
input {
    http {
        id => "web-logger"
        port => 5045
        codec => json
        response_headers => {
            "Access-Control-Allow-Origin" => "*"
            "Content-Type" => "text/plain"
            "Access-Control-Allow-Headers" => "Origin, Content-Type"
        }
        tags => ["web-logger"]
    }
    ...
}

filter {
    date {
        match => [ "timestamp", "UNIX_MS" ]
        target => "@timestamp"
    }

    if ("web-logger" in [tags]) {
        useragent {
            source => "useragent"
            target => "useragent"
        }

        mutate {
            add_field => { "[@metadata][index]" => "web-logger-%{+yyyy.MM.dd}" }
        }
    }
}

output {
    elasticsearch {
        hosts => "elasticsearch:9200"
        index => "%{[@metadata][index]}"
    }
}
```

Obr. 23: Konfiguračný súbor nástroja Logstash

Ukladanie všetkých log záznamov do jedného Elasticsearch indexu by mohlo spôsobiť v budúcnosti problémy s rýchlosťou pri načítavaní a vyhľadávaní. Aby sa predišlo tejto situácií, pri ukladaní záznamov je vytvorený jeden index pre každý deň. Týmto sa zabezpečí existencia viacerých menších indexov. Nové záznamy sú vkladané vždy iba do najnovšieho indexu, a všetky ostatné indexy slúžia iba na čítanie. Taktiež sa zjednoduší údržba a čistenie starých, už nepotrebných záznamov. Môže sa odstrániť celý index bez obáv z toho, že by sa narušili novšie dáta. Názov indexu kam sa má spracovávaný záznam uložiť je definovaný v bloku *filter*. Pomocou funkcie *mutate* a *add_field* je nastavený názov indexu vo formáte „web-logger-YYYY.MM.DD“.

5.4. Agent pre webové aplikácie

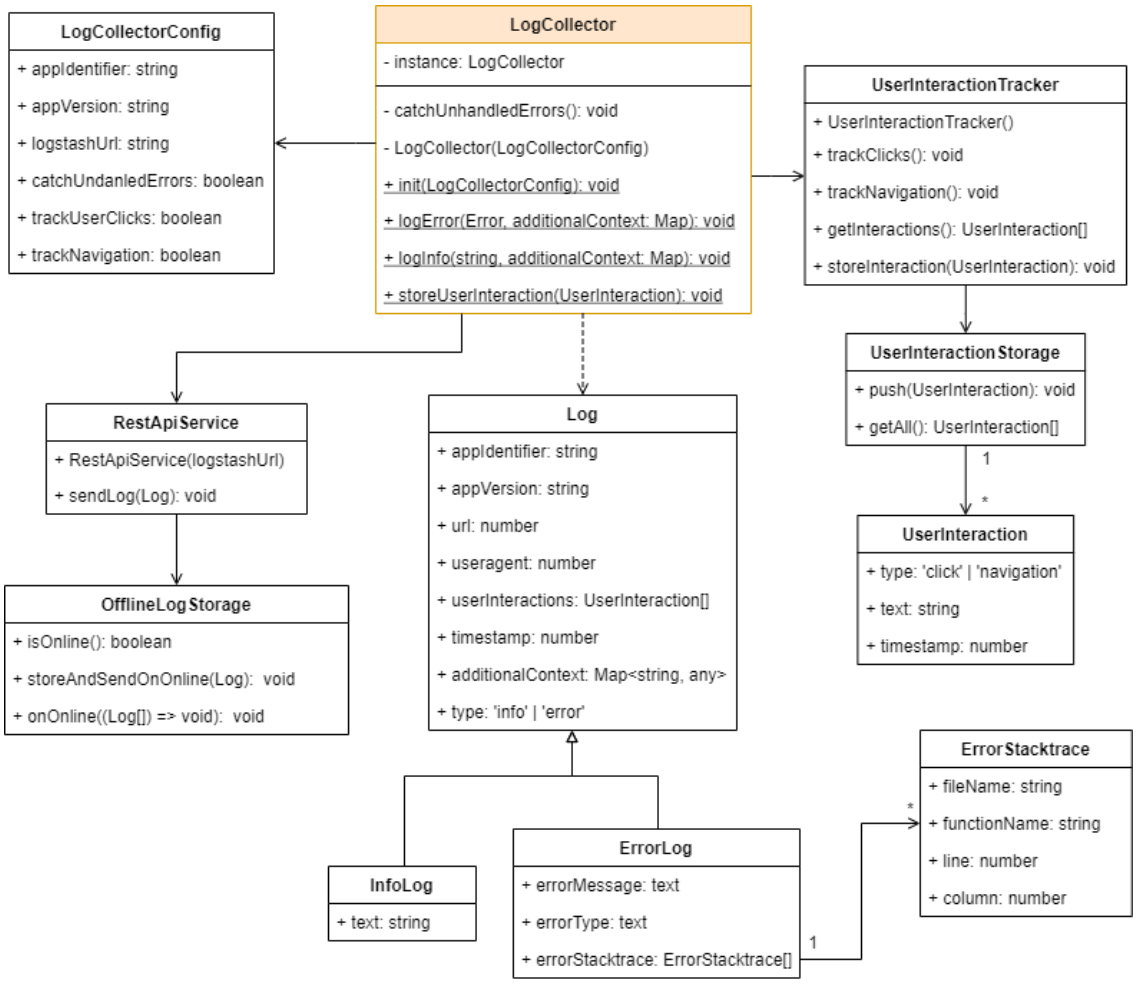
V predchádzajúcej kapitole bol pomocou nástroja Logstash vytvorený prístupový bodu, ktorý dokáže prijať a spracovať hlásenia o udalosti vo formáte JSON. Ďalším dôležitým krokom je vytvorenie knižnice, ktorá bude zaznamenávať chybové a iné udalosti vo webových aplikáciách, a následne ich bude posielať do spomínaného nástroja Logstash. Pri návrhu tejto knižnice je dôležité zamerať sa na nasledovné požiadavky:

- Knižnica by mala byť schopná zachytiť hlásenie o chybe a zaznamenať informácie ako text chyby, typ chyby a miesto výskytu v zdrojovom kóde.
- Záznam by mal obsahovať informácie o zariadení, na ktorom chyba vznikla. Tieto údaje by mali obsahovať hlavne typ a verziu prehliadača, prípadne typ zariadenia a operačný systém, na ktorom webová aplikácia beží.
- Knižnica môže poskytovať informácie o slede činností, ktoré používateľ vykonal, a ktoré viedli ku vzniku chyby. Zaznamenávanie týchto aktivít by malo byť voliteľné.
- Knižnica by mala byť jednoducho implementovateľná do akejkoľvek webovej aplikácie, a hlave do aplikácií vytvorených pomocou knižníc React a Angular.
- Knižnica by mala byť schopná zaznamenať a uchovať hlásenie o chybe aj v prípade, že aplikácia beží bez pripojenia na internet. Po obnovení pripojenia by malo byť hlásenie odoslané na server.
- Okrem chybového hlásenia by mala byť knižnica schopná odoslať aj informatívne hlásenie, ktoré obsahuje všetky prvky chybového hlásenia, okrem údajov týkajúcich sa chyby.

Knižnica nesie názov *log-collector-web-core*, a na Obr. 24 na ďalšej strane je pomocou diagramu tried vyobrazená jej štruktúra. Trieda *LogCollector* je hlavnou a jedinou triedou, ktorá sa používa v monitorovanej aplikácií. Pri návrhu tejto triedy bol kladený dôraz na jej jednoduché a praktické použitie. Trieda poskytuje rozhranie pomocou statických metód, ktoré slúžia na inicializáciu knižnice a následné ukladanie záznamov o udalosti. Forma implementácie pomocou statických metód bola zvolená z dôvodu jednoduchšieho použitia knižnice, keďže nie je nutné posielať inštanciu triedy medzi komponentami webovej aplikácie. Na implementáciu triedy *LogCollector* bol použitý návrhový vzor jedináčik, aby bolo zaručené, že v monitorovanej aplikácií existuje iba jedna inštancia tejto triedy.

Trieda *RestApiService* sa stará o odosielanie záznamov do nástroja Logstash pomocou metódy *sendLog*. Pri svojom inicializovaní vyžaduje adresu nástroja Logstash ako parameter v konštruktore. V spolupráci s triedou *OfflineLogStorage* zabezpečuje uchovanie záznamov v prípade absencie internetového pripojenia. Ak webová aplikácia beží dočasne bez internetového pripojenia, log záznamy sa uchovajú do úložiska poskytovaného prehliadačom. Po obnovení pripojenia sa uložené záznamy odošlú do nástroja Logstash.

LocalStorage je úložisko, ktoré je účasťou Web Storage špecifikácie. Web Storage je špecifikácia W3C, ktorá poskytuje funkcie na ukladanie údajov na strane klienta v prehliadači. Toto úložisko je výkonnejšie ako tradičné súbory cookie a ľahšie sa s ním pracuje. [29] Pomocou *LocalStorage* je možné uchovať neodoslané log záznamy aj po zatvorení webovej stránky.



Obr. 24: Štruktúra knižnice log-collector-web-core

Knižnica taktiež poskytuje možnosť zaznamenávania interakcie používateľa s webovou stránkou. Toto je dosiahnuté automatickým zaznamenávaním kliknutí myšou na elementy na stránke a zaznamenaním navigácie medzi stránkami. Zaznamenanie všetkých kliknutí na webovej stránke je dosiahnuté preťažením metódy *addEventListener* na triede *EventTarget.* Triedu *EventTarget* poskytuje DOM a slúži na riadenie udalostí, ktoré sa na webovej stránke vyskytnú. Zachytené udalosti sú následne uložené v úložisku *UserInteractionStorage*, kde čakajú do momentu, kedy sú pripojené k log záznamu a odosielané do Logstash nástroja.

Na nasledujúcom obrázku je demonštrované inicializácia a použitie knižnice.

```
LogCollector.init({
    appIdentifier: 'test-app',
    appVersion: require('../package.json').version,
    logstashUrl: 'http://localhost:5045',
});

try {
    ...
} catch (error) {
    LogCollector.logError(error)
}
```

Obr. 25: Ukážka použitia agenta pre webové aplikácie

Knižnica je inicializovaná pomocou statickej metódy *init* na triede *LogCollector.* Táto metóda prijíma ako parameter objekt, s nasledovnými atribútmi:

Názov atribútu	**Predvolená hodnota**	**Povinný atribút**	**Popis**
appIdentifier	--	Áno	Identifikátor webovej aplikácie. Identifikátor by mal byť unikátny pre každú monitorovanú aplikáciu. Príklad: 'lcm-app-frontend'.
appVersion	--	Áno	Verzia webovej aplikácie.
logstashUrl	--	Áno	Adresa servera Logstash, kde sa posielajú všetky záznamy
catchUnhandledErrors	true	Nie	Atribút, ktorý určuje, či má knižnica automaticky zachytávať chybové udalosti, ktoré neboli ošetrené *try-catch* blokmi v mieste vzniku.
catchUserClicks	true	Nie	Určuje, či sa zaznamenávajú kliknutia myšou.
catchNavigation	true	Nie	Určuje, či sa zaznamenáva sled navigácií medzi stránkami.

Tabuľka 4: Zoznam konfiguračných parametrov knižnice pre webové aplikácie

5.4.1. Agent pre knižnicu React

Na predchádzajúcich stranách bola popísaná knižnica *log-collector-web-core*, ktorá môže byť použitá v akejkoľvek webovej aplikácií. React webová aplikácia vyžaduje mierne odlišný prístup ku zachyteniu chybovej udalosti a zaznamenaniu navigácie medzi stránkami. Na uľahčenie integrácie s React aplikáciou je vytvorená samostatná knižnica, ktorá využíva funkcionalitu poskytovanú knižnicou *log-collector-web-core*.

Ak vznikne chyba v časti používateľského rozhrania, nemala by pokaziť celú aplikáciu. Na vyriešenie tohto problému ponúka React koncept ohraničenia chyby, tzv. *Error Boundaries*. React poskytuje možnosť vytvoriť komponent, ktorý zachytáva chyby v kóde kdekoľvek vo svojom strome komponentov. V prípade vzniku chyby je táto chyba zaznamenaná a namiesto pôvodného stromu komponentov sa zobrazí komponent, ktorý definuje programátor. Tieto komponenty zachytávajú chyby počas vykresľovania, v metódach životného cyklu a v konštruktoroch celého stromu pod nimi. [30]

Pre jednoduché využitie konceptu ohraničenia chyby je vytvorený komponent *ErrorBoundary.* Pri vzniku chyby je na tomto komponente zavolaná statická metóda *componentDidCatch,* ktorá ako parameter prijíma samotný objekt chyby. V tejto metóde sa následne volá metóda *logError* na triede *LogCollector* z knižnice *log-collector-web-core.*

Navigácia medzi stránkami je ďalšou špecifickou črtou React aplikácie. Pri bežnej webovej stránke sa pri prechode medzi stránkami načítava znovu celý obsah. Okamih, kedy je načítaný celý obsah webovej stránky môžeme využiť na vytvorenie záznamu, že došlo k prechodu z jednej stránky na inú. Toto však nie je možne pri React aplikácií, keďže sa pri prechode zo stránky na stránku nenačítava znovu celý obsah stránky.

Knižnica *react-router-dom* sa používaná na spravovanie navigácie v React aplikácií. [31] Poskytuje rozhranie, pomocou ktorého je možné zaznamenať prechod medzi stránkami. Na nasledujúcom obrázku je ukážka kódu komponentu, ktorý zaznamenáva navigáciu v React aplikácií pomocou knižnice *react-router-dom,* a následné o tejto udalosti informuje nástroj *LogCollector* z knižnice *log-collector-web-core.*

```
import { useHistory } from 'react-router-dom';

export const NavigationTracker = (): React.FunctionComponent => {
    useHistory().listen((location, action) => {
        LogCollector.storeUserInteraction({
            type: 'navigation',
            timestamp: Date.now(),
            text: location.pathname
        });
    })
    return null;
}
```

Obr. 26: Komponent na zaznamenávanie navigácie v React aplikácií

5.4.2. Agent pre knižnicu Angular

Podobne ako v prípade knižnice React, aj aplikačný rámec Angular poskytuje vlastné mechanizmy na zaznamenávanie chýb a navigácie na stránke. Ako bolo popísane v kapitole 3.3.2, Angular poskytuje mechanizmus vkladania závislosti (Dependency Injection). Všetky služby a komponenty musia byť definované v konfigurácií modulu. Každá Angular aplikácia má jeden koreňový modul, v ktorom sa nastavujú služby pre celú aplikáciu. Práve toto miesto je vhodné na nastavenie centrálneho monitorovania chýb v aplikácií a aj zaznamenávanie navigácie medzi stránkami.

Podobne ako pre knižnicu React, tak aj pre Angular je vytvorená samostatná knižnica, ktorá využíva funkcionalitu knižnice *log-collector-web-core.* Hlavným účelom tejto knižnice je zjednodušiť integráciu knižnice s Angular aplikáciou. Knižnica obsahuje dve služby:

- *LogCollectorErrorHandler* – predstavuje službu na automatické zaznamenávanie chybových hlásení.

- *NavigationTrackerService* – táto služba slúži na zaznamenávanie prechodov medzi stránkami. Zaznamenávanie kliknutí na stránke nebolo potrebné v prípade knižnice Angular meniť a využíva sa pôvodné automatické zaznamenávanie z knižnice *log-collector-web-core.*

Trieda *LogCollectorErrorHandler* implementuje rozhranie *ErrorHandler* z knižnice Angular. Toto rozhranie vyžaduje implementáciu verejnej metódy *handleError,* ktorá prijíma chybový objekt ako parameter v prípade, že v aplikácií vznikla chyba. V tejto metóde sa chyba spracuje zavolaním statickej metódy *logError* na triede *LogCollector.* Na zjednodušenie vytvárania inštancie triedy *LogCollectorErrorHandler* slúži továrenská metóda *createErrorHandler,* ktorá prijíma ako parameter objekt s konfiguračnými parametrami knižnice. Konfigurácia knižnice pre Angular ostala v porovnaní so základnou knižnicou nezmenená.

Služba *NavigationTrackerService* využíva funkcionalitu poskytovanú inštanciou triedy *Router* z knižnice Angular. Atribút *events* na inštancii triedy *Router* predstavuje prúd všetkých udalostí spojených s navigáciou v aplikácií. Odfiltrovaním nežiadúcich udalostí dostávame iba udalosti navigácie medzi stránkami, a tie následne spracujeme metódou *storeUserInteraction* na triede *LogCollector.* Spôsob inicializácie služby *NavigationTrackerService* ako aj *LogCollectorErrorHandler* je možné vidieť na nasledujúcom obrázku.

```
@NgModule({
  ...
  providers: [
    {
      provide: ErrorHandler,
      useValue: createErrorHandler({
        appIdentifier: require('../../package.json').name,
        appVersion: require('../../package.json').version,
        logstashUrl: 'http://localhost:5045'
      })
    },
    {
      provide: NavigationTrackerService,
      deps: [Router],
    }
  ]
})
export class AppModule { }
```

Obr. 27: Konfigurácia agenta pre Angular aplikáciu

5.5. Aplikácia na správu log záznamov

V predchádzajúcich kapitolách boli nastavené nástroje platformy Elastic Stack. Nastroj Logstash poskytuje prístupový bod, pomocou ktorého prijíma hlásenia o udalostiach a následne ich ukladá v databáze Elasticsearch. V ďalšej kapitole bola popísaná knižnica, ktorá zachytáva a posiela hlásenia o udalostiach do nástroja Logstash. V tejto kapitole sa zaoberám serverovou aplikáciou, ktorá implementuje funkcionalitu, ktorá nie je poskytnutá platformou Elastic Stack. Pri návrhu tejto aplikácie som sa zameral na naplnenie nasledovných požiadaviek:

- Aplikácia by mala poskytovať prístup k uložených hláseniam o udalostiach.
- Pomocou tejto aplikácie by malo byť možné spravovať monitorované aplikácie a ich verzie

- Aplikácia by mala byť schopná prijať a uložiť súbory s mapovaním produkčného JavaScript kódu webovej aplikácie
- Aplikácia by mala umožňovať prepojenie hlásenia o udalosti so systémom na sledovanie incidentov (angl. Issue Tracker)

Pri návrhu aplikácie na správu log záznamov boli zvážené dve možné architektúry. Prvou možnosťou bolo vytvorenie aplikácie, ktorá sa spúšťa priamo z operačného systému používateľa. Aplikácia by priamo komunikovala s databázou. Výhodou tohto riešenia je rozdelenie záťaže na viaceré počítače. Veľkou nevýhodou však je komplikovanejšia distribúcia nových vydaní a nutnosť udržiavania databázu v rôznych verziách.

Druhou možnosťou je použiť architektúru klient-server a nasadiť aplikáciu na server. Aplikácia na serveri by komunikovala s databázou a obsluhovala by požiadavky všetkých klientov. Klientom by v tomto prípade bola webová aplikácia. Keďže by sa jednalo o webovú aplikáciu, oproti prvej metóde by odpadla nutnosť inštalácie na klientskych počítačoch. Ďalšou výhodou je jednoduchá a rýchla distribúcia nových vydaní aplikácie. Jednou z nevýhod je kumulácia všetkých výpočtových požiadaviek klientov na jedno miesto. V prípade tejto aplikácie sa však nepredpokladá, že aplikáciu bude používať veľké množstvo používateľov, ktorí by vykonávali náročné operácie súčasne. Na základe týchto poznatkov bola zvolená architektúra typu klient-server.

Klientská časť predstavuje webovú aplikáciu, ktorá zobrazuje dáta a poskytuje rozhranie na nastavenie celého nástroja. Táto časť je popísaná v kapitole 5.6.

Serverová časť aplikácie je vytvorená v jazyku Java a pomocou aplikačného rámca Spring Framework, ktorým sa zaoberám v prvej podkapitole. Následne je popísaná celková štruktúra aplikácie a jej jednotlivé časti. V kapitole 5.5.3 sa zaoberám integráciou so systémami na sledovanie incidentov a v poslednej podkapitole triedením uložených záznamov pomocou neurónových sietí.

5.5.1. Spring Framework

Spring je aplikačný rámec s otvoreným zdrojovým kódom, ktorý vytvoril Rod Johnson a ktorý opísal vo svojej knihe. Bol vytvorený za účelom riešenia zložitosti vývoja podnikových aplikácií. [32]

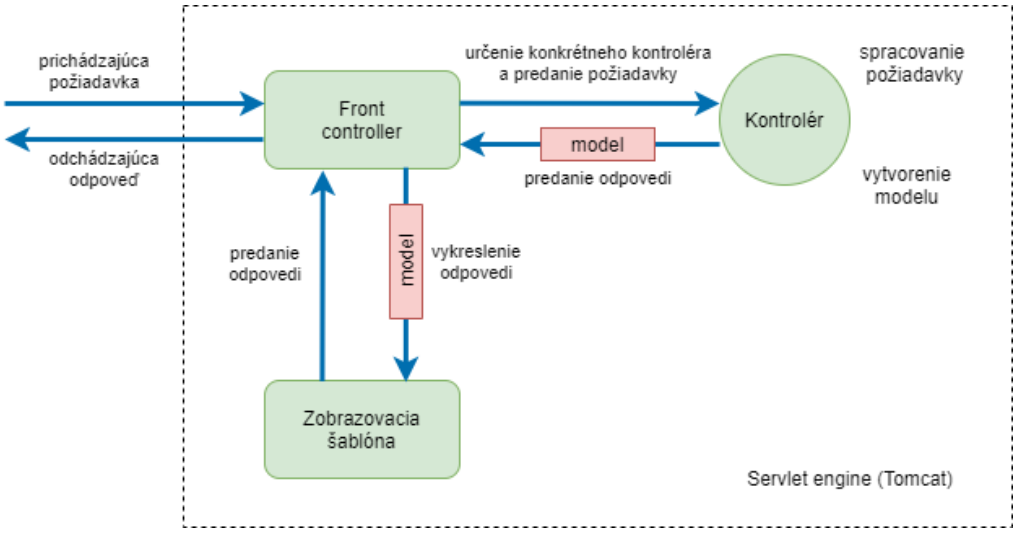
Spring charakterizujú nasledovné vlastnosti:

- Spring podporuje voľné spojenie medzi závislosťami pomocou techniky inverzie riadenia (z angl. Inversion of Control, IoC). Pri použití IoC sú objektom poskytnuté ich závislosti namiesto toho, aby si tieto závislosti pre seba vytvárali samy.
- Spring prináša podporou aspektovo orientovaného programovania, ktoré umožňuje oddelenie aplikačnej logiky od systémových služieb (ako sú auditovanie a správa transakcií). Aplikačné objekty robia to, čo majú robiť - vykonávajú aplikačnú logiku - a nič viac. Nemusia sa zaujímať o systémové procesy, ako napríklad logovanie alebo transakčná podpora.
- Spring umožňuje konfigurovať a skladať zložité aplikácie z jednoduchších komponentov. Poskytuje taktiež bohatú infraštruktúru (správa transakcií, rámce na ukladanie dát atď.), takže programátor sa môže plne sústrediť na vývoj aplikačnej logiky. [32]

Spring Web MVC

Jedným z použitých Spring modulov je Spring Web MVC (model-view-controller). Tento modul je navrhnutý okolo centrálneho komponentu *DispatcherServlet*, ktorý smeruje jednotlivé dopyty ku konkrétnym kontrolérom na spracovanie. Kontrolér je označený anotáciou *@Controller* a *@RequestMapping*. Spring MVC taktiež umožňuje vytvárať REST API rozhranie pomocou anotácie *@RestController, @PathVariable* a ďalších. [33]

DispatcherServlet je implementácia návrhového vzoru *Front Controller.* Tento návrhový vzor využíva aj mnoho ďalších aplikačných rámcov. Všetky dopyty na server sú prijaté jedným vstupným bodom, tzv. Front Controller. Tento kontrolér určí pomocou triedy *HandlerMapping,* ktorý kontrolér je určený na spracovanie daného dopytu, a predá mu dopyt na spracovanie. Po spracovaní dopytu vráti kontrolér odpoveď s modelom, ktorý je pomocou zobrazovacej šablóny spracovaný. Následne je odpoveď odoslaná klientovi, ktorý vytvoril daný dopyt na server. Na obrázku je vyobrazený princíp fungovania Spring MVC.

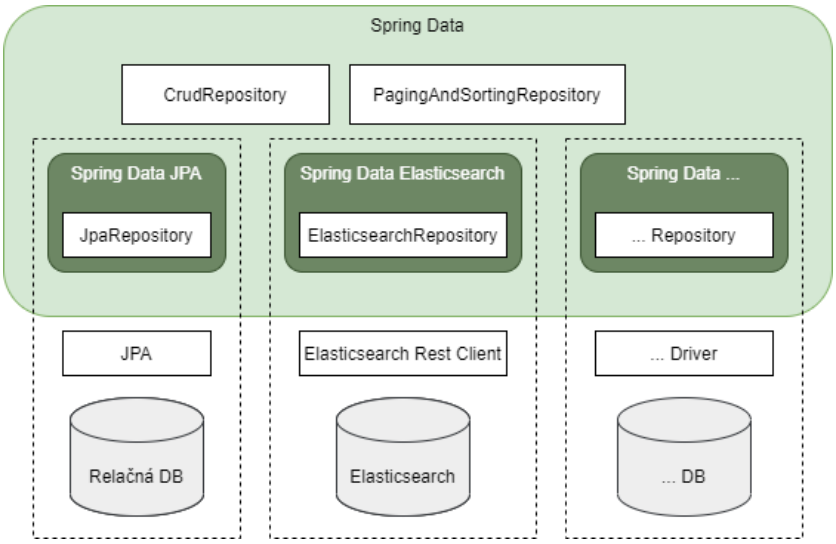


Obr. 28: Princíp fungovania Spring MVC [33]

Spring Data

Spring Data je označenie pre rodinu modulov pre prácu s rôznymi databázami. Spring Data poskytuje jednotný prístup k rôznym zdrojom dát, vrátane relačných a objektových databáz. V tejto práci je použitý modul Spring Data JPA na prácu s relačnou databázou, a Spring Data Elasticsearch na prístup k dátam v Elasticsearch databáze.

Spring Data JPA je súčasť väčšej rodiny Spring Data. Cieľom Spring Data JPA je výrazne zjednodušiť implementáciu vrstiev aplikácie, ktoré pristupujú k dátam v databáze. Vývojár definuje rozhranie jednotlivých tried, pomocou ktorých aplikácia pristupuje k dátam. Spring následne zabezpečí ich implementáciu. [34]



Obr. 29: Štruktúra Spring Data modulov [35]

Základnými prvkami Spring Data sú repozitáre a doménové triedy, ktoré modelujú uložené dáta. Aby bola trieda rozpoznaná aplikačným rámcom ako repozitár, používa sa anotácia *@Repository*. Následne je nutné definovať rozhranie, ktoré repozitár poskytuje. Modul Spring Data JPA poskytuje tri možnosti, ako definovať databázový dopyt:

- Vygenerovaním dopytu z názvu metódy, napríklad: *findById*
- Pomocou anotácie *@Query* v repozitári
- Pomocou anotácie *@NamedQuery* v doménovej triede

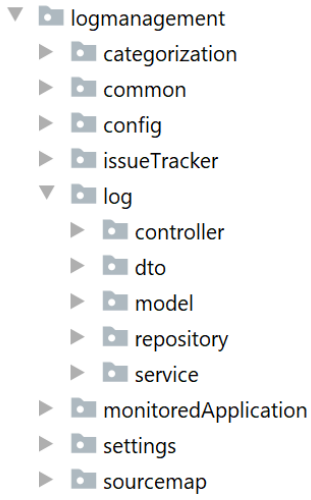
Spring Data Elasticsearch implementuje podobné princípy ako Spring Data JPA.

Doménové objekty sa často nazývajú aj entity. Pri relačnej databáze je entita spravidla reprezentovaná databázovou tabuľkou. Inštancia tejto triedy zase reprezentuje jeden záznam v tabuľke. Každá trieda entity musí spĺňať nasledovné požiadavky:

- Na označenie entity musí byť použitá anotácia *@Entity* z balíka *javax.persistance*.
- Trieda musí obsahovať jeden bezparametrický konštruktor s verejnou alebo chránenou viditeľnosťou.
- Trieda a metódy nesmú byť deklarované ako finálne.
- Entita môže dediť z inej triedy.
- Atribúty musia byť deklarované ako neverejné, a môžu byť prístupné iba cez metódy, tzv. gettery a settery. [36]

5.5.2. Implementácia systému

V predchádzajúcej kapitole bol predstavený aplikačný rámec Spring, spolu s modulmi, ktoré sú v tejto práci využité. Celá aplikácia je rozdelená do niekoľkých balíkov. Každý balík zastrešuje určitú funkcionalitu aplikácie, no niektoré balíky sú spoločné pre celú aplikáciu. Na nasledujúcom obrázku je vidieť rozdelenie aplikácie do balíkov.



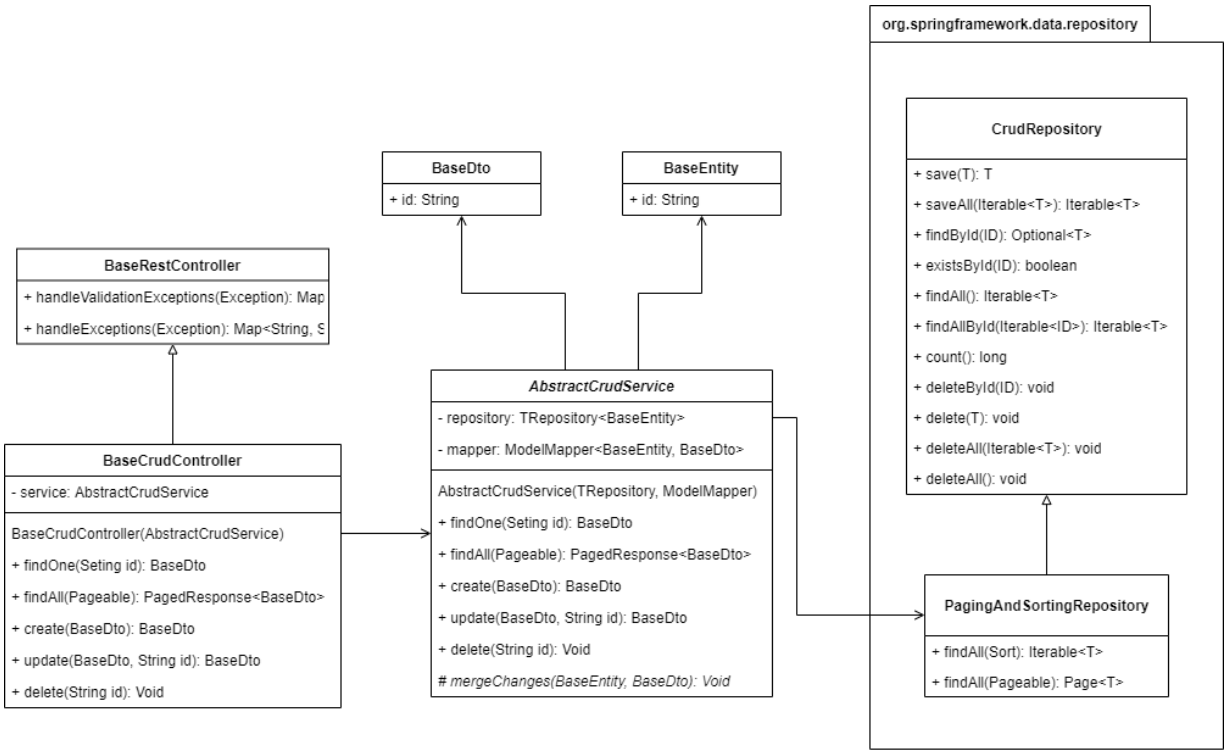
Obr. 30: Štruktúra balíkov v aplikácií

Balík *logmanagement.common* a *logmanagement.config*

Balík *logmanagement.common* a *logmanagement.config* obsahujú spoločnú funkcionalitu pre celú aplikáciu. V balíku *logmanagement.config* sú umiestnené všetky konfiguračné triedy pre Spring Framework. Balík *logmanagement.common* obsahuje prevažne triedy, ktoré implementujú spoločnú funkcionalitu a v ostatných balíkoch je táto funkcionalita zdedená. Jedná sa hlavne o triedy:

- *BaseRestController* – jednou z funkcií tohto kontroléra je spracovávanie výnimiek, ktoré môžu vzniknúť v potomkoch tejto triedy. Týmto je zabezpečený centralizovaný prístup k výnimkám, a teda chybová odpoveď servera bude vždy konzistentná.
- *BaseCrudController* – implementuje takzvané CRUD metódy, teda metódy na vytváranie, aktualizovanie, vymazanie a načítanie objektu.
- *AbstractCrudService* – implementuje CRUD metódy, ktoré sú použité v *BaseCrudController* kontroléri.
- *BaseEntity* a *BaseDto* – základné triedy, od ktorých dedia všetky ostatné entity a DTO objekty spoločné atribúty.

Na nasledujúcom obrázku je pomocou UML diagramu tried znázornená štruktúra balíka *logmanagement.common*. Pre zachovanie čitateľnosti sú znázornené iba najpodstatnejšie triedy. Na diagrame je vidieť všetky metódy, ktoré dané triedy poskytujú.



Obr. 31: Štruktúra balíka logmanagement.common

BaseCrudController poskytuje prístupové body na základnú manipuláciu s uloženými dátami pomocou nasledovných prístupových bodov:

- `GET /{id:string}` – slúži na načítanie jedného objektu na základe jeho identifikátora
- `GET /` – slúži na načítanie viacerých objektov. Tento prístupový bod je stránkovaný. Stránkovanie sa nastavuje pomocou nasledovných parametrov v URL:
  - *page:int* – definuje číslo stránky. Východzia hodnota je 0
  - *size:int* – definuje množstvo objektov na jednej stránke. Východzia hodnota je 10.
  - *sort:<atribút, zoradenie>* – určuje, podľa akého atribútu sú objekty zoradené a v akom smere. Príklad: *sort=name,asc*
- `POST /` – slúži na vytvorenie nového objektu. Dáta objektu sú posielané v tele HTTP dopytu.
- `PUT /{id:string}` – slúži na aktualizovanie objektu s daným identifikátorom na nové hodnoty, ktoré sú posielané v tele HTTP dopytu.
- `DELETE /{id:string}` – slúži na odstránenie objektu s daným identifikátorom.

Balík *logmanagement.monitoredApplication*

Pri monitorovaní viacerých aplikácií je potrebné mať o nich prehľad. Aplikácia je identifikovaná jedinečným identifikátorom, ktorý je rovnaký ako identifikátor obsiahnutý v zázname o udalostí. Pretože sa aplikácie môžu časom vyvíjať, nestačí poznať iba ich názov, ale je potrebné poznať aj ich verziu. Práve správu týchto informácií zabezpečuje tento balík.

Pre každú aplikáciu je možné nastaviť nasledujúce atribúty:

- Unikátny identifikátor aplikácie, ktorý je nemenný.
- Názov aplikácie v užívateľsky príjemnej forme. Tento atribút je voliteľný.
- Identifikátor projektu v systéme na správu incidentov. Tento atribút spája konkrétnu monitorovanú aplikáciu s projektom v externom nástroji, napr. v nástroji GitLab. Implementácia tohto prepojenia je popísaná v jednej z nasledujúcich kapitol.
- Zoznam verzií aplikácie.

Vydanie novej verzie aplikácie je s najväčšou pravdepodobnosťou spojené so zmenou zdrojového kódu, preto je potrebné pre každú verziu nahrať súbory mapujúce produkčný zdrojový kód do pôvodného, teda súbory Source Map.

Vytvorenie novej verzie a nahranie súborov s mapovaním zdrojového kódu je aktivita, ktorá je nevyhnutná pri každom vydaní aplikácie. Je teda viac ako žiaduce túto aktivitu automatizovať a to zahrnutím do procesu zostavovania aplikácie. Pre tieto účely slúžia nasledovné prístupové body:

- `POST /monitoredApplicationVersion/cli` – prístupový bod na vytvorenie novej verzie aplikácie. Očakáva nasledujúce atribúty v tele HTTP požiadavky:
    - *applicationIdentifier* – identifikátor aplikácie. Príklad: 'my-super-application'
    - *applicationVersion* – verzia naposledy zostavenej aplikácie. Príklad: '1.0.5'
    - *gitIdentifier* – identifikátor pre *Git commit,* v ktorom sa nachádza repozitár počas zostavenia novej verzie. Tento atribút je voliteľný.
- `POST /sourcemap` – prístupový bod na nahratie súborov *Source Map*. Telo požiadavky musí obsahovať nasledujúce atribúty:
    - *applicationIdentifier* – identifikátor aplikácie, ku ktorej patria nahrané súbory
    - *applicationVersion* – verzia aplikácie, ku ktorej patria nahrané súbory
    - *files* – súbory obsahujúce mapovanie zdrojového kódu

Pri vytváraní novej verzie aplikácie sa atribút *gitIdentifier* momentálne nepoužíva. Jeho existencia však umožňuje v budúcnosti implementovať rozšírenie aplikácie, ktoré by prepojilo Git úložisko so záznamom udalostí.

Balík *logmanagement.log*

Záznamy o udalostiach sú ukladané do databázy pomocou nástroja Logstash a Elasticsearch. Aby sme mohli používateľovi uložené záznamy zobraziť, je ich nutné z databázy načítať a pomocou REST API poskytnúť používateľskému rozhraniu. Celá implementácia sa nachádza v balíku *logmanagement.log*.

Pomocou kontroléra *LogController* je vytvorené nasledovné API:

- `GET /search` - slúži na načítanie zoznamu záznamov o udalostiach na základe vstupných parametrov:
    - *dateFrom* – časová známka, ktorá určuje aké najstaršie záznamy o udalostiach sa majú načítať. Hodnota je v milisekundách.
    - *dateTo* – časová známka, ktorá určuje aké najnovšie záznamy o udalostiach sa majú načítať. Hodnota je v milisekundách.
    - parametre stránkovanie sú rovnaké ako v prípade `BaseCrudController` kontroléra.

- `GET /histogram` - slúži na načítanie dát pre histogram, v časovom rozmedzí definovanom parametrami:
    - *dateFrom*, *dateTo* – parametre definujú časové rozmedzie, pre ktoré má byť skonštruovaný objekt s dátami pre histogram. Hodnoty sú v milisekundách.
- `GET /{logId:string}` - slúži na načítanie jedného záznamu o udalosti na základe poskytnutého identifikátora.

LogController využíva službu *LogService*, ktorá poskytuje jednotlivé dáta pre kontrolér. Na komunikáciu s databázou je použitý spomínaný Spring Data Elasticsearch modul. Konfigurácia Spring Data Elasticsearch modulu sa nachádza v súbore ElasticsearchConfiguration v balíku *logmanagement.config*. Konfigurácia spočíva v implementovaní rozhrania *AbstractElasticsearchConfiguration* a implementovaním abstraktnej metódy *elasticsearchClient()*, ktorá vracia inštanciu triedy *RestHighLevelClient.*

Pri konštruovaní vyhľadávacieho dopytu je možné postupovať rôznymi spôsobmi. Dokumentácia samotného Spring Data Elasticsearch je rozsiahla, bohatá na príklady a programátor tam dokáže nájsť to, čo potrebuje. V príklade na nasledujúcom obrázku najprv vytvoríme *Criteria* objekt, ktorý obsahuje obmedzenia, ktorými chceme limitovať načítanú množinu záznamov. V tomto príklade sa jedná o obmedzenie na časový výskyt záznamu v určitom časovom intervale. Následne je tento objekt použitý na vytvorenie vyhľadávacieho dopytu, ktorý môže obsahovať niekoľko podmienok reprezentovaných inštanciou triedy *Criteria*. Posledným krokom je zavolanie metódy *search* na inštancii triedy *ElasticsearchOperations* zo Spring Data Elasticsearch modulu. Metóda očakáva tri parametre: vyhľadávací dopyt, triedu, na ktorú sa mapujú dáta z databázy a názov indexu, v ktorom chceme vyhľadávať. Po úspešnom načítaní dát je potrebné jednotlivé záznamy transformovať na DTO objekty, ktoré sú následne odoslané v odpovedi na vyhľadávací dopyt.

```
// LogController
// vytvorenie kriterií vyhladávania
Criteria criteria = new Criteria("@timestamp")
        .between(dateFrom, dateTo);
service.search(criteria, pageable);

// LogService
public PagedResponse<LogDto> search(Criteria criteria, Pageable pageable) {
    // zkonstruovanie vyhladavacej query
    Query query = new CriteriaQuery(criteria, pageable);

    // použitie Spring Data Elasticsearch na vyhladanie objektov
    SearchHits<LogRecord> hits = elasticsearchOperations
            .search(query, LogRecord.class, IndexCoordinates.of(LOGS_INDEX));

    // transformácia LogRecord na LogDto objekt
    List<LogDto> objects = hits.get()
            .map(SearchHit::getContent)
            .map(LogMapper::toDto)
            .collect(Collectors.toList());
    return new PagedResponse<>(objects, hits.getTotalHits(), pageable);
}
```

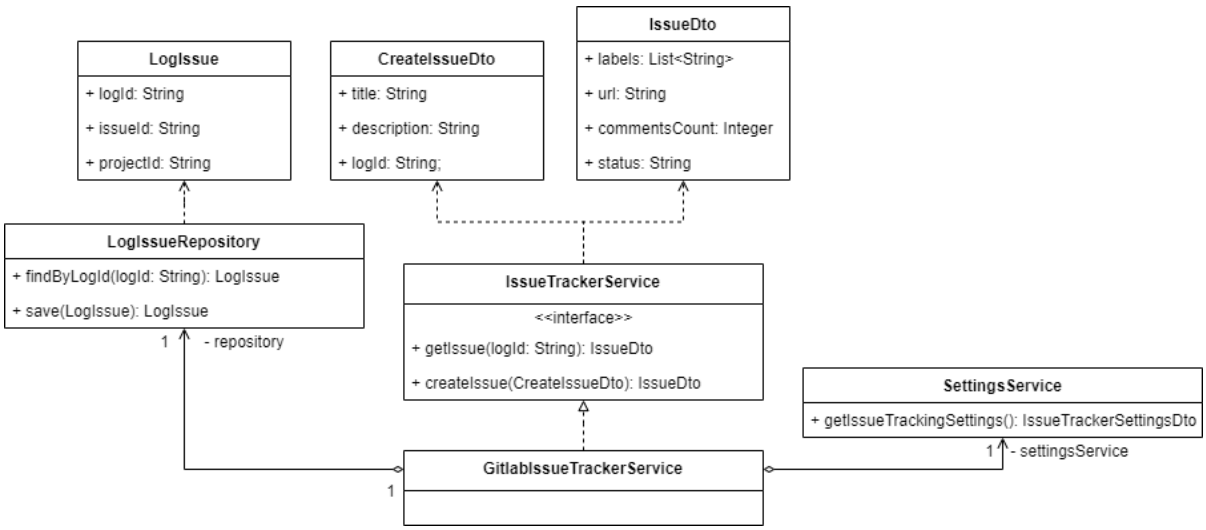
Obr. 32: Použitie Spring Data Elasticsearch

5.5.3. Integrácia so systémami na sledovanie incidentov

Jednou z ďalších požiadaviek na systém je schopnosť spolupracovať s nástrojom na sledovanie incidentov. Prepojenie chybových hlásení s incidentmi zlepšuje proces integrovanie nástroja s existujúcimi procesmi a nástrojmi v organizáciách. Primárnym cieľom bolo vytvoriť riešenie na integráciu s populárnym nástrojom GitLab.

GitLab je webový nástroj, ktorý pomáha počas celého životného cyklu softvéru. Poskytuje správcu úložiska Git, funkciu wiki dokumentácie, sledovanie incidentov a nástroje na zostavovanie, testovanie a nasadzovanie vydaní aplikácií. GitLab je do veľkej miery podobný s platformou GitHub, avšak je viac využívaný vo firemnom prostredí, hlavne kvôli možnosti inštalácie nástroja na vlastnom serveri.

V tejto kapitole sa zaoberám iba prepojením vyvíjanej aplikácie s časťou na sledovanie incidentov. GitLab poskytuje rozsiahle REST API, pomocou ktorého je možné nástroj ovládať. V rámci prieskumu možností komunikácie s nástrojom GitLab bolo zistené, že existuje Java knižnica, ktorá uľahčuje používanie zmieneného REST API. Knižnica sa volá GitLab4J API.



Obr. 33: Diagram tried pre balík logmanagement.issueTracker

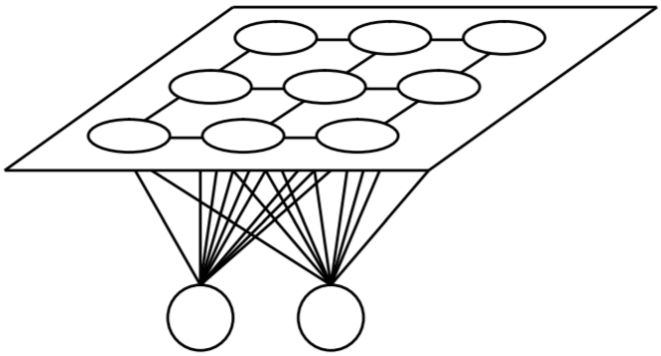
Na inicializáciu knižnice je potrebná adresa GitLab servera a prístupový kľúč, ktorý je vygenerovaný v nastaveniach nástroja GitLab. Tie informácie sú uložené v nastaveniach monitorovanej aplikácie a sú prístupné pomocou metódy *getIssueTrackingSettings()* na triede *SettingService*. Pomocou knižnice GitLab4J je vytvorený incident v GitLab nástroji. Následne sa jeho identifikátor uloží s identifikátorom log záznamu do databázy, aby bolo následne možné spojiť GitLab incident a log záznam. Pre vytvorenie GitLab incidentu je potrebné nastaviť minimálne titulok a popis incidentu, ktorý v tomto prípade obsahuje údaje o chybovej udalosti. Na rýchlejšie prepojenie obsahuje popis incidentu aj URL, na ktorej je chybový záznam dostupný. Predvolený titulok a popis incidentu je možné upraviť pri vytváraní prepojenia v používateľskom rozhraní.

5.5.4. Triedenie záznamov pomocou neurónových sietí

Pre spojenie podobných záznamov do skupín je využitá metóda zhlukovania, konkrétne algoritmus Kohonenových máp. V tejto podkapitole popíšem samotný princíp algoritmu a jeho implementáciu.

Kohonenová mapa je typ neurónovej siete, ktorej hlavnou ideou je nájsť priestorovú reprezentáciu zložitých dátových štruktúr. Inými slovami, aby triedy si podobných vektorov, boli reprezentované neuróny blízkymi si v danej topológii. Táto vlastnosť je typická aj pre skutočný mozog, kde napríklad jeden koniec sluchovej časti mozgovej kôry reaguje na nízke frekvencie, zatiaľ čo opačný koniec reaguje na frekvencie vysoké. Týmto spôsobom je možné mnohodimenzionálne údaje zobraziť v jednoduchšom priestore. Najčastejšie použitie Kohonenovho algoritmu je vo forme dvojdimenzionálnej implementácie. [37]

Topológia takej siete vyzerá nasledovne:



Obr. 34: Ukážka Kohonenovej mapy [37]

Svojou schopnosťou samoorganizácie a zhlukovania dát s podobnými vlastnosťami do skupín sú Kohonenové mapy priamo predurčené pre aplikácie určené na rozhodovanie, rozlišovanie a triedenie objektov, signálov, značiek a pod. Častou sférou aplikácie je rozpoznávanie reči, napríklad prepis hovoreného slova na text. Výhodou SOM je, že o spracovávaných dátach nie je teoreticky nutné čokoľvek vedieť a algoritmus si ich sám preberie a roztriedi. Možné aplikácie:

- spracovanie reči
- úprava zvuku
- spracovanie obrazu (videa a fotiek)
- hľadanie a detekcia osôb podľa fotografií
- bezpečnostné aplikácie
- prepis ručne písaného textu
- hľadanie podobných znakov v úplne neznámych signáloch
- odstránenie neznámeho šumu
- automatické triedenie [38]

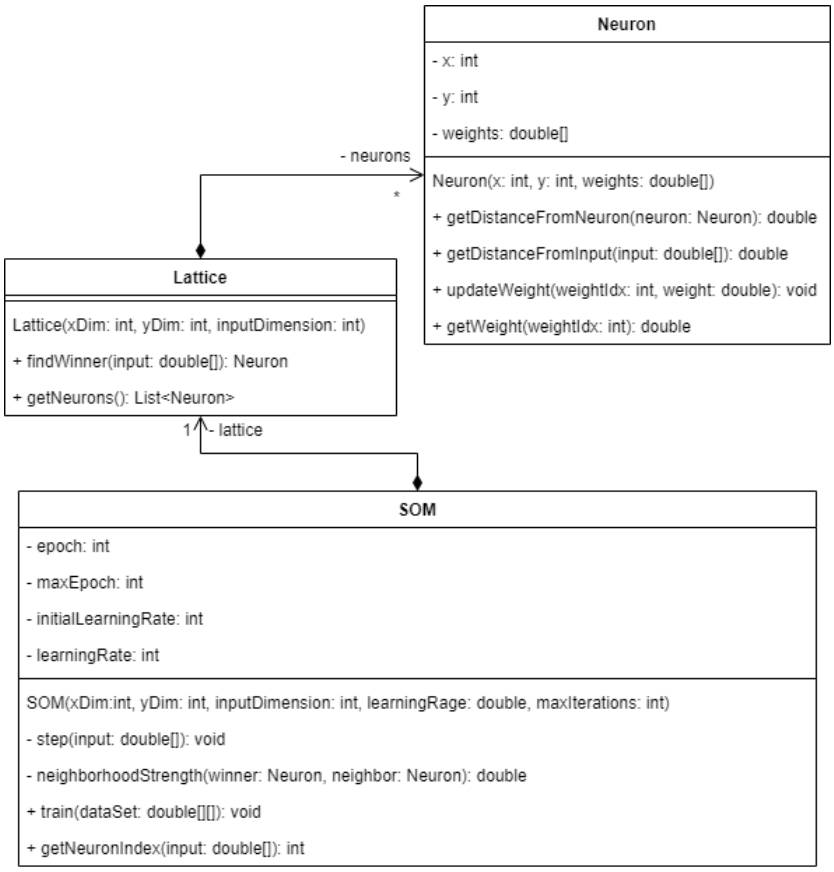
Topologická štruktúra určuje, ktoré neuróny spolu v sieti susedia. Pre adaptačný proces je tiež dôležité zaviesť pojem *okolia J* výstupného neurónu (*j**) s polomerom *R*, čo je množina všetkých neurónov ($j \in J$), ktorých vzdialenosť v sieti je od daného neurónu (*j**) menšia alebo rovná *R:* $J = \{j; d(j,j^*) \leq R\}$. $d(j,j^*)$ predstavuje vzdialenosť dvoch neurónov *j* a *j**.

Samotný Kohonenov algoritmus je nasledovný:

1. Inicializuj sieť: inicializuj váhy w_{ij} malými náhodnými číslami. Nastav priemer susedstva *R* na maximum tak, aby pokrýval celú výstupnú vrstvu
2. Predlož vstup vo forme $x_0(t),\ x_1(t), x_2(t), \dots, x_{n-1}(t)$, kde $x_i(t)$ je vstup uzlu *i* v čase *t.*
3. Vypočítaj vzdialenosť medzi vstupným vektorom a každým neurónom $j\ (j = 1, \dots, m)$ pomocou vzťahu $d_j = \sum_i (w_{ij}(t) - x_i(t))^2$
4. Nájdi víťazný neurón pomocou minima d_j a označiť ho j^*
5. Aktualizuj váhy pre neurón *j** a jeho susedy $j \in J(j^*)$. Nové váhy sú dané vzťahom: $w_{ij}(t+1) = w_{ij}(t) + \alpha\left[x_i(t) - w_{ij}(t)\right]$
6. Zmenši parameter učenia *α* a polomer topologického susedstva *R* s cieľom stabilizovať váhy a lokalizovať maximálne aktivity.
7. Choď na krok 2. [37]

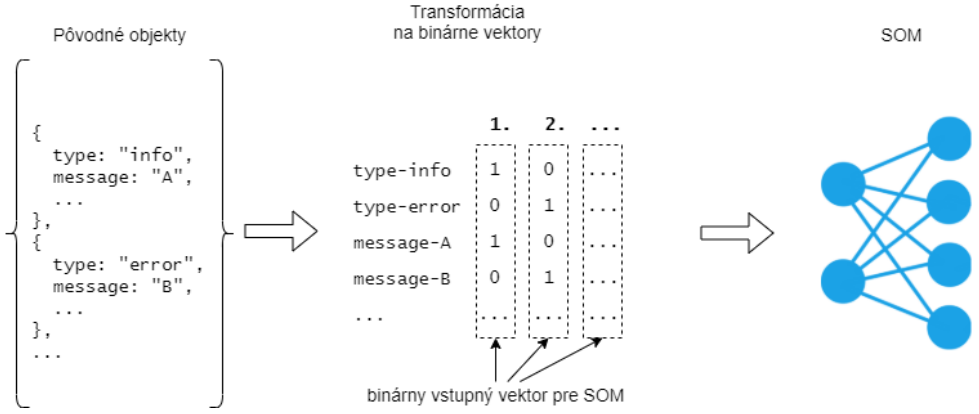
Implementácia

Algoritmus s obslužným kódom na nachádza v balíku *logmanagement.categorization*. Trieda *SOM* je hlavnou triedou implementácie. Je zodpovedná za inicializáciu algoritmu, samotné učenie a následne vyvolanie informácie, ku ktorému neurónu daný vstup inklinuje. Inštancia triedy *Lattice* obsahuje zoznam neurónom výstupnej vrstvy a metódy, na nájdenie najlepšieho (víťazného) neurónu pre daný vstup. Trieda *Neuron* je poslednou triedou implementácie tohto algoritmu. Atribúty *x* a *y* predstavujú polohou neurónu v mriežke, a atribút *weights* obsahuje váhy, spojené s neurónom. Trieda *Neuron* poskytuje metódy na nastavovanie váh, a metódy na získanie euklidovskej vzdialenosti od iného neurónu resp. vstupného vektora. Vzdialenosť jedného neurónu od druhého sa využíva na určenie množiny susedov víťazného neurónu počas trénovania siete. Samotný proces učenia je implementáciou vyššie popísaného algoritmu. Na nasledujúcom Obr. 35 na ďalšej strane je diagram tried, ktorý zachytáva štruktúru modelu SOM.



Obr. 35: Diagram tried implementácie SOM modelu

Aby sme mohli triediť objekty v databáze, najprv ich je potrebné upraviť na formu, ktorú SOM dokáže spracovať. Prvá z podmienok aby SOM fungovala správne je, aby jednotlivé vstupné vektory do algoritmu mali rovnaký počet prvkov. Ďalším obmedzením je, že SOM požaduje na vstupe vektory s číselnými hodnotami. Preto je nutné objekty na takéto vektory transformovať. Proces transformácie je pre lepšie pochopenie znázornený na Obr. 36. Jeho podstata spočíva vo vytvorení dvojíc skladajúcich sa z názvu atribútu a jeho hodnoty. Následne sa určí, ktoré z týchto dvojíc sú zastúpene v objekte, a do výsledného vektoru sa na toto miesto nastaví hodnota 1. Výsledný vektor tak má binárnu podobu.



Obr. 36: Transformácia vstupných dát pre SOM

5.6. Používateľské prostredie

Táto kapitola je venovaná poslednému chýbajúcemu článku celého systému. Jedná sa používateľské rozhranie na ovládanie a používanie celej aplikácie. Táto časť je nemenej dôležitá, pretože je samotnému používateľovi najbližšie.

Na tvorbu používateľského prostredia je použitá knižnica React vo svojej najnovšej verzií 17. Používateľské prostredie je do veľkej miery skladané z vizuálnych komponentov knižnice Material-UI. Táto knižnica poskytuje množstvo predpripravených komponentov, ako napríklad tlačidla, menu lišty, ikony a mnohé iné.

Projekt aplikácie je rozdelený na tri hlavné priečinky:

- *components* – v tomto priečinku sa nachádzajú menšie komponenty, ktoré sa opakujú na rôznych stránkach. Nachádza sa tu napríklad komponent na vykreslenie menu, alebo aj komponent na vykreslenie tabuľky. O týchto komponentoch sa hovorí, že sú hlúpe, lebo iba vykresľujú poskytnuté dáta a neobsahujú žiadnu logiku. Kvôli tomu môžu byť použité na rôznych stránkach.
- *containers* – v tomto priečinku sa nachádzajú komponenty reprezentujúce jednotlivé stránky aplikácie, ako napríklad detail záznamu udalosti alebo stránku s nastaveniami. Na rozdiel od prvej skupiny komponentov obsahujú tieto komponenty logiku aplikácie a sú zodpovedné za načítanie dát zo servera.
- *rest* – v tomto priečinku sa nachádzajú pomocné funkcie, ktoré sa využívajú pri komunikácií so serverom pomocou REST API.

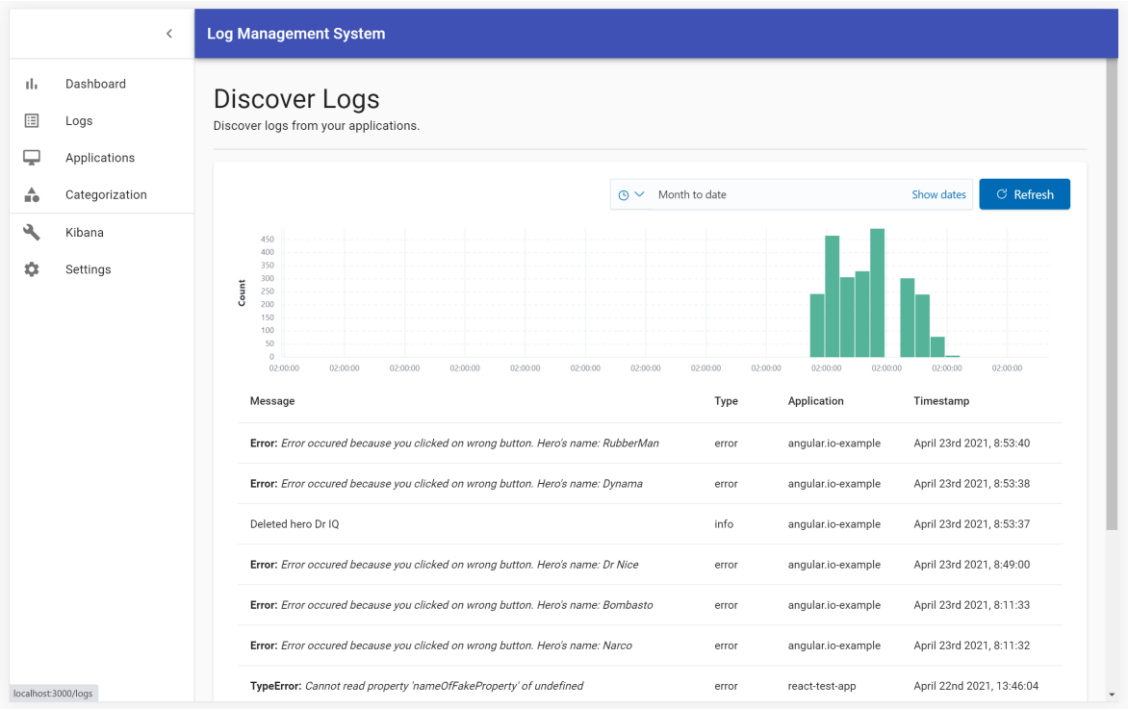
Používateľské rozhranie sa skladá zo šiestich hlavných stránok: *Dashboard, Logs, Applications, Categorization, Kibana* a *Settings.*

Stránka *Dashboard* poskytuje rýchly prehľad o aktuálnej situácií monitorovaných aplikácií. Nástroj Kibana ponúka širokú škálu grafov. Tieto grafy môžu byť spojené do ucelených prehľadov a následné zdieľané pomocou vygenerovanej URL adresy. Stránka *Dashboard* zobrazuje práve jeden takýto prehľad.

Základná konfigurácia nástroja sa nachádza na stránke *Settings.* Používateľ tu môže nastaviť:

- adresu GitLab servera
- prístupový kľúč na komunikáciu s GitLab serverom
- prístupovú URL adresu pre nástroj Kibana
- URL adresu *dashboard* stránky z nástroja Kibana

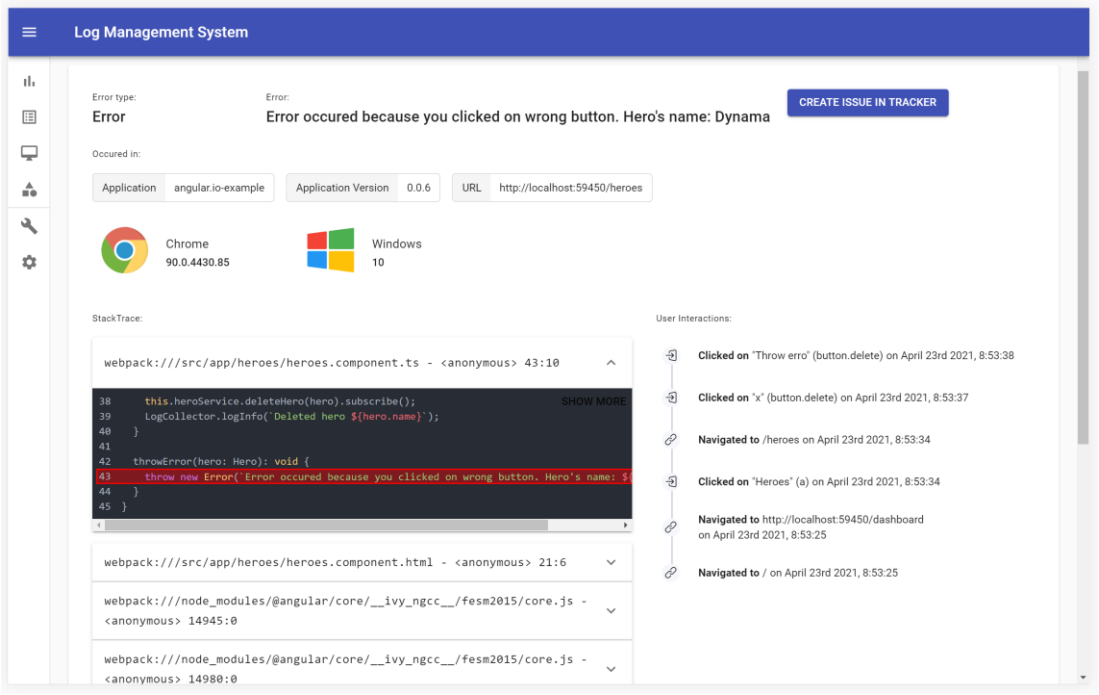
Jednou z najdôležitejších stránok je zobrazovanie záznamov udalostí. Túto stránku je možné vidieť na nasledujúcom obrázku. Obsahuje zoznam načítaných udalostí s výpisom najdôležitejších údajov a to text udalosti, identifikátor aplikácie, v ktorej udalosť nastala a čas výskytu. Graf nad zoznam pomáha získať vizuálny prehľad o priebehu udalostí. Posledným prvkom je ovládač na nastavenie časového intervalu, pre ktorý chceme záznamy o udalostiach vyhľadať.



Obr. 37: Používateľské prostredie - zoznam log záznamov

Zoznam log záznamov neposkytuje používateľovi detailné informácie o kontexte udalosti, a tak je vytvorená stránka detailu. Tu je možné vidieť na Obr. 38 na ďalšej strane. Stránka detailu obsahuje nasledovné informácie:

- Typ udalosti a jej text.
- Identifikačné údaje aplikácie, v ktorej daná udalosť vznikla. Tieto údaje obsahujú identifikátor aplikácie, jej verziu a URL adresu, na ktorej daná udalosť vznikla.
- Informácie o prehliadači a operačnom systéme, na ktorom bežala webová aplikácia.
- V prípade záznamu chybovej udalosti je zobrazený zdrojový kód z miesta, kde sa vyskytla chyba. Zdrojový kód je čitateľný vďaka súborom Source Map, pomocou ktorých je zdrojový kód v minimalizovanej podobe prevedený naspäť do čitateľnej podoby.
- Napravo od komponentu so zdrojovým kódom sa nachádza zoznam činnosťami, ktoré viedli ku vzniku zaznamenanej udalosti.
- V pravom hornom rohu sa nachádza tlačidlo na vytvorenie prepojenia s nástrojom na sledovanie incidentov. Toto tlačidlo je zobrazené iba v prípade, že pre daný záznam nebol vytvorený incident. Ak takýto incident existuje, je zobrazený odkaz na daný incident, spolu so základnými informáciami ako napr. počet komentárov.



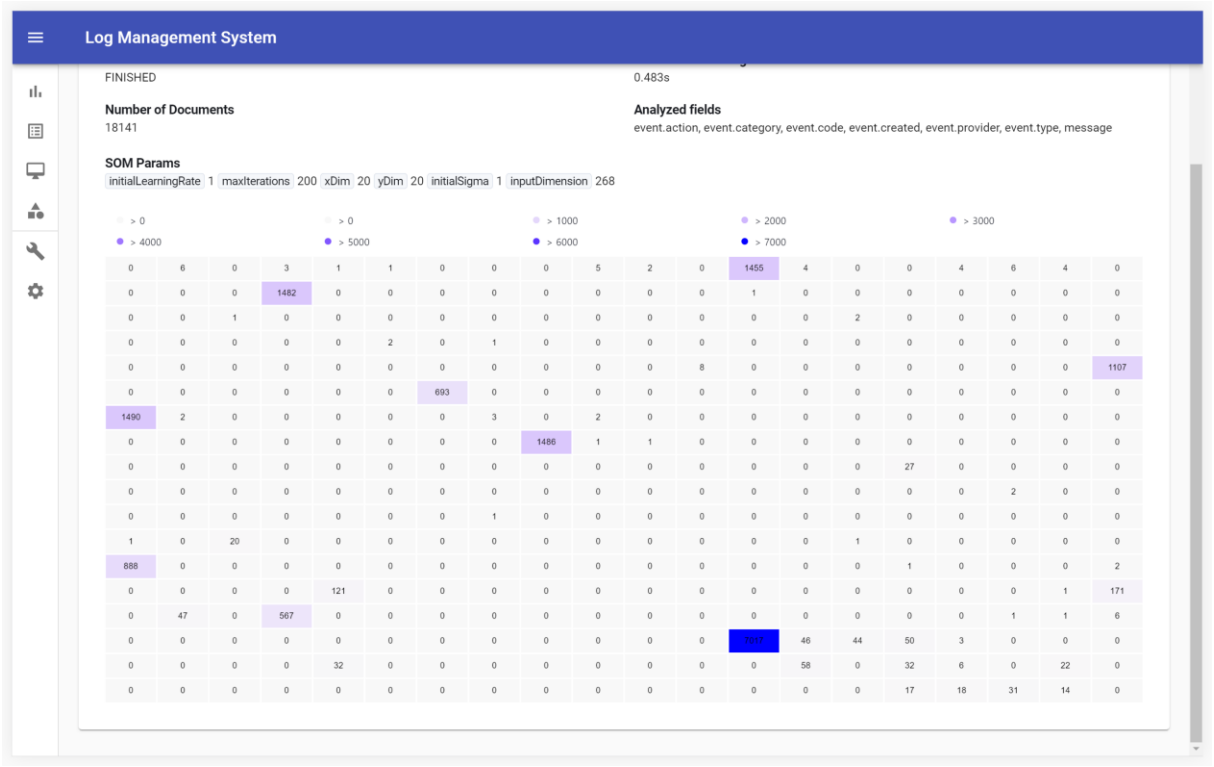
Obr. 38: Používateľské prostredie - detail log záznamu

Na stránke *Categorization* sa nachádza zoznam ukončených, ale aj prebiehajúcich procesov triedenia objektov. Z tohto miesta je možné vytvoriť zadanie pre triedenie objektov, ktoré musí obsahovať nasledovné parametre:

1. V prvom kroku sa zobrazí zoznam dostupných indexov v databáze Elasticsearch a počet dokumentov, ktoré obsahujú. Je možné vybrať jeden alebo skupinu indexov, ktoré budú predmetom triedenia pomocou algoritmu Kohonenových máp.
2. Po zvolení indexu je nutné vybrať atribúty, na základe ktorých bude množina objektov roztriedená. Je nutné vybrať aspoň jeden atribút.

Detail zadania spolu s výsledkami je zobrazený na Obr. 39 na ďalšej strane. Na tejto stránke sú zobrazené nasledovné údaje:

- Index v databáze Elasticsearch, z ktorého pochádzajú analyzované dokumenty
- Trvanie procesu učenia a triedenia celej množiny dokumentov
- Počet roztriedených dokumentov
- Zoznam parametrov, na základe ktorých boli dokumenty triedené
- Maticu reprezentujúcu neuróny Kohonenovej mapy a počet dokumentov inklinujúcich k jednotlivým neurónom.
- Po kliknutí na jednotlivé neuróny sa zobrazí vzorka dokumentov, ktoré inklinujú k danému neurónu. Týmto je možné overiť, či algoritmus roztriedil dokumenty správne.



Obr. 39: Používateľské prostredie - detail triedenia dokumentov

Záver

Cieľom tejto práce bolo vytvoriť nástroj na zaznamenávanie chybových hlásení vo webových aplikáciách, ktorý by zjednodušil ich údržbu. Teoretická časť práce sa zaoberá problematikou zaznamenávania udalostí v aplikáciách. Sú tu popísané dôvody, prečo je vhodné, aby aplikácia generovala záznamy o udalostiach, čo by taký záznam mal obsahovať a k čomu ho je možné použiť.

Vyvinutý nástroj je určený hlavne na monitorovanie webových aplikácií a preto v ďalšej kapitole popisujem ich základné vlastnosti a najpoužívanejšie technológie, ktoré sú používané na ich vývoj. Je tu predstavený jazyk JavaScript, jeho silné ale aj slabé stránky. V neposlednom rade popisujem knižnice na tvorbu webových aplikácií, ako sú ReactJS a Angular.

Jedným z cieľov tejto práce bolo preskúmať a porovnať existujúce nástroje na správu chybových hlásení. Tomu sa venuje celá štvrtá kapitola, ktorá predstavuje nástroje Sentry, Rollbar a platformu Elastic Stack. Sentry a Rollbar sú veľmi populárne nástroje, ale poskytujú iba základné funkcie v bezplatnom balíku. Pri skúmaní platformy Elastic Stack bolo zistené, že môže slúžiť ako dobrý základ pre implementáciu vlastného riešenia.

Piata kapitola predstavuje praktickú časť práce. Opisuje proces návrhu a prijaté rozhodnutia. Samotný nástroj využíva infraštruktúru platformy Elastic Stack, ktorej inštalácia a nastavenie je popísané v samostatnej kapitole. Pre jednoduchosť sa Elastic Stack konfiguruje pomocou virtualizačného nástroja Docker. Ďalej sa zaoberám vytvorením knižnice *log-collector-web-core* pre webové aplikácie a JavaScript. Okrem zachytávania chybových hlásení táto knižnica umožňuje zaznamenávať aj udalosti informatívneho charakteru. Pre lepšie pochopenie kontextu chyby knižnica zaznamenáva aktivitu používateľa na webových stránkach. Pre jednoduchú integráciu s knižnicami React a Angular boli vytvorené ďalšie dve knižnice, ktoré implementujú špecifické vlastnosti každej technológie.

V nasledujúcej podkapitole je vytvorená serverová časť nástroja pomocou aplikačného rámca Spring. Táto kapitola popisuje štruktúru projektu a jeho jednotlivé časti. Ďalej je navrhnutá integrácia s nástrojmi na správu incidentov, čo bol jeden z cieľov tejto práce. Nad rámec zadania obsahuje práca aj implementáciu neurónových sietí, za účelom triedenia uložených záznamov. Pre proces triedenia sa používa algoritmus Kohonenových máp a je možné triediť všetky dokumenty uložené v databáze Elasticsearch. Výsledky sú potom graficky prezentované. Poslednou časťou kapitoly o návrhu a implementácii je vytvorenie používateľského rozhrania pomocou knižnice React.

Počas tvorby tejto diplomovej práce sa vo firme, v ktorej mal byť vytvorený nástroj použitý, zmenilo vedenie. To viedlo k veľkým organizačným zmenám a k môjmu odchodu z tejto firmy. S poľutovaním musím konštatovať, že vytvorený nástroj je síce funkčný, ale nikdy nebol nasadený do produkčného prostredia, pre ktoré bol vytvorený. Samotný vývoj a testovanie nástroja prebehlo iba na vzorových aplikáciách, ktoré simulovali výskyt chýb.

Architektúra vytvoreného nástroja umožňuje implementovať v budúcnosti nové rozšírenia. Nástroj môže byť rozšírený o monitorovanie sieťovej aktivity vo webovej aplikácií. Takáto informácia by pomohla odhaliť chybu, ktorá vznikla kvôli zlému internetovému pripojeniu alebo nesprávnemu formátu dát prijatých zo servera. Pri ukladaní ďalších informácií o používateľovi a jeho aktivite na

webových stránkach, by bolo potrebné zaoberať sa témou ochrany osobných údajov a dodržiavania štandardov GDPR.

Literatúra

1. Chuvakin, Anton, Schmidt, Kevin and Phillips, Chris. *Logging and Log Management.* s.l. : Syngress, 2013. ISBN: 978-1-59749-635-3.

2. *World Wide Web Consortium (W3C).* [Online] [cit. 25. 04 2021.] Dostupné na: https://www.w3.org/.

3. Progressive web apps (PWAs). *MDN Web Docs.* [Online] [cit. 06. 03 2021.] Dostupné na: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps.

4. What are Progressive Web Apps? [Online] [cit. 06. 03 2021.] Dostupné na: https://web.dev/what-are-pwas/.

5. Flanagan, David. *JavaScript: The Definitive Guide.* 6th edition. Sebastopol : O'Reilly, 2011. s. 1096. ISBN: 978-0-59680-552-4.

6. Dynamic programming language. *MDN Web Docs.* [Online] [cit. 06. 03 2021.] Dostupné na: https://developer.mozilla.org/en-US/docs/Glossary/Dynamic_programming_language.

7. TypeScript for the New Programmer. *TypeScript Docs.* [Online] [cit. 12. 04 2021.] Dostupné na: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html.

8. Stack Overflow Trends. [Online] [cit. 15. 03 2020.] Dostupné na: https://insights.stackoverflow.com/trends?tags=reactjs%2Cangular%2Cvue.js%2Cjquery%2Cbackbone.js%2Cember.js%2Cknockout.js.

9. Alex Banks, Eve Porcello. *Learning React.* Sebastopol : O'Reilly Media, Inc., 2017. ISBN: 978-1-491-95462-1.

10. What is Angular? *Angular Docs.* [Online] [cit. 15. 04 2021.] Dostupné na: https://angular.io/guide/what-is-angular.

11. Angular History. *NgDevelop.* [Online] [cit. 15. 04 2021.] Dostupné na: https://www.ngdevelop.tech/angular/history/.

12. Code minification. *Webplatform.* [Online] [cit. 01. 04 2021.] Dostupné na: https://webplatform.github.io/docs/concepts/programming/javascript/minification/.

13. Seddon, Ryan. Introduction to JavaScript Source Maps. *HTML5 Rocks.* [Online] 21. 03 2012. [cit. 01. 04 2021.] Dostupné na: https://www.html5rocks.com/en/tutorials/developertools/sourcemaps/.

14. Application Monitoring and Error Tracking Software. *Sentry.* [Online] [cit. 15. 04 2021.] Dostupné na: https://sentry.io/welcome/.

15. Sentry Pricing. [Online] [cit. 05. 04 2021.] Dostupné na: https://sentry.io/pricing/.

16. Self-Hosted Sentry. *Sentry.* [Online] [cit. 03. 04 2021.] Dostupné na: https://develop.sentry.dev/self-hosted/.

17. Error Monitoring and Crash Reporting Software. *Rollbar.* [Online] [cit. 15. 04 2021.] Dostupné na: https://rollbar.com/product/.

18. Rollbar Pricing Plans. [Online] [cit. 08. 04 2021.] Dostupné na: https://rollbar.com/pricing/.

19. Rollbar - Error Tracking Software. [Online] [cit. 08. 04 2021.] Dostupné na: https://rollbar.com/product/monitor.

20. Clinton Gormley, Zachary Tong. *Elasticsearch: The Definitive Guide.* s.l. : O'Reilly Media, 2015. ISBN: 978-1-449-35854-9.

21. Logstash: Collect, Parse, Transform Logs. [Online] [cit. 06 03 2021.] Dostupné na: https://www.elastic.co/logstash.

22. Beats: Data Shippers for Elasticsearch. *Elastic.* [Online] [cit. 03. 04 2021.] Dostupné na: https://www.elastic.co/beats/.

23. Kibana: Explore, Visualize, Discover Data. [Online] [cit. 06. 03 2021.] Dostupné na: https://www.elastic.co/what-is/kibana.

24. The Unified Modeling Language. [Online] [cit. 07. 04 2021.] Dostupné na: https://www.uml-diagrams.org/.

25. Docker overview. *Docker Docs.* [Online] [cit. 05. 04 2021.] Dostupné na: https://docs.docker.com/get-started/overview/.

26. What is a Container? *Docker.* [Online] [cit. 05. 04 2021.] Dostupné na: https://www.docker.com/resources/what-container.

27. Bootstrapping a cluster. *Elastic Docs.* [Online] [cit. 08. 04 2021.] Dostupné na: https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-discovery-bootstrap-cluster.html.

28. Networking. *Elastic Docs.* [Online] [cit. 08. 04 2021.] Dostupné na: https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-network.html.

29. Dixit, Shwetank. Web Storage: Easier, More Powerful Client-Side Data Storage. [Online] 05. 03 2013. [cit. 15. 04 2021.] Dostupné na: https://dev.opera.com/articles/web-storage/.

30. Error Boundaries. *React Docs.* [Online] [cit. 15. 04 2021.] Dostupné na: https://reactjs.org/docs/error-boundaries.html.

31. Declarative Routing for React.js. *React Router.* [Online] [cit. 20. 04 2021.] Dostupné na: https://reactrouter.com/web/guides/quick-start.

32. WALLS, CRAIG. *Spring in Action.* Fifth Edition. New York : Manning Publications, 2019. s. 520. ISBN: 978-1-61729-494-5.

33. Web MVC framework. *Spring Docs.* [Online] [cit. 15. 04 2021.] Dostupné na: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html.

34. Spring Data JPA. *Spring Docs.* [Online] [cit. 15. 04 2021.] Dostupné na: https://spring.io/projects/spring-data-jpa.

35. Spring Data – One API To Rule Them All. [Online] [cit. 20. 04 2021.] Dostupné na: https://www.infoq.com/articles/spring-data-intro/.

36. Requirements for Entity Classes. *Oracle Docs.* [Online] [cit. 15. 04 2021.] Dostupné na: https://docs.oracle.com/cd/E19798-01/821-1841/bnbqb/index.html.

37. Vondrák, Ivo. *Neuronové sítě.* Ostrava : Fakulta elektrotechniky a informatiky VŠB. Učební text., 2009.

38. Vojáček, Antonín. Samoučící se neuronová síť - SOM, Kohonenovy mapy. [Online] 14. 05 2006. [cit. 20. 04 2021.] Dostupné na: https://www.kiv.zcu.cz/studies/predmety/uir/NS/Samouc_NN2.pdf.